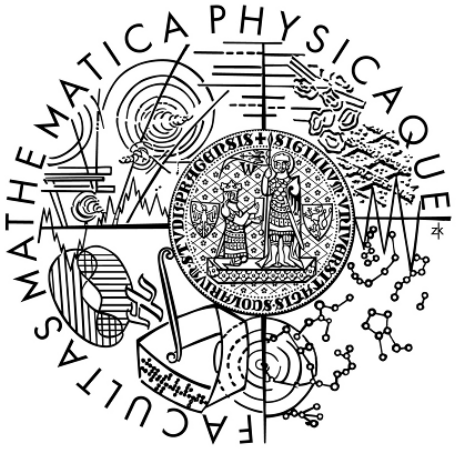
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Bc. Pavel Baštecký

Data Modeling for Static Analysis of Web Applications

Department of Distributed and Dependable Systems

Supervisor: Mgr. David Hauzar, Ph.D.

Study programme: Informatics

Branch of study: Software systems

Prague 2015

I would like to thank my supervisor Mgr. David Hauzar, Ph.D. and RNDr. Jan Kofroň, Ph.D. for their leadership and helpful advices. I appreciate that David wasn't stopped helping me even when he left faculty after his Ph. D. graduation. Thank you.

Many thanks to my friends and colleagues from Weverca software project: Matyáš Brenner, Marcel Kikta, David Škorvaga and Miroslav Vodolán. Weverca would never exit without your team work.

Special thanks to my parents: mom Dagmar Baštecká and dad Zdenek Baštecký. Thank you for supporting me all the time. I wouldn't be here without your help, I will never forget that. I thank my sister, her husband and their children for a little spark of insanity which helped me survive all these years.

Thank you all.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V ..... dne ..... podpis

Název práce:	Data Modeling for Static Analysis of Web Applications
Autor:	Bc. Pavel Baštecký
Katedra:	Katedra distribuovaných a spolehlivých systémů
Vedoucí bakalářské práce:	Mgr. David Hauzar, Ph.D.
Klíčová slova:	PHP, statická analýza, paměťový model, weverca

Abstrakt

PHP je velmi oblíbený jazyk, často používaný na implementaci serverové části webových aplikací. Jazyk je velmi jednoduchý na používání a i proto je na celém internetu velké množství menších stránek, ale i rozsáhlejších aplikací, napsaných v jazyce PHP. Velká obliba PHP však způsobuje, že mnoho lidí vyhledává jeho slabiny s cílem narušit bezpečnost webových aplikací.

Weverca je první nástroj schopný provést komplexní bezpečnostní analýzu celé stránky napsané v moderní verzi PHP a vyhledat informace o možných bezpečnostních rizicích aplikace. Výkon nástroje Weverca je však omezen časovou a paměťovou náročností, která je způsobena neefektivitou reprezentace paměti PHP stránky.

Cílem této práce je nalézt a vyřešit hlavní nedostatky původní implementace paměťového modelu. Výsledkem je nová implementace, která minimalizuje nároky původního řešení.

Title:	Data Modeling for Static Analysis of Web Applications
Author:	Bc. Pavel Baštecký
Department:	Department of Distributed and Dependable Systems
Supervisor:	Mgr. David Hauzar, Ph.D.
Keywords:	PHP, static analysis, memory model, weverca

Abstract

The PHP is a very popular language which is used to write a server side part of web applications. The language is very simple to use and there are lots of small or more complex pages across the internet. But the great widespread of the PHP attracts the people which want to harm and compromise security of the web applications.

The weverca analyzer is the first tool which is able to perform complex security analysis of a full page written in the modern version of the PHP and give information about possible security risks in the application. But the performance of Weverca is limited by its time and memory complexity caused by inefficient inner representation of a PHP memory state.

The goal of this thesis is to find and solve main problems of the original memory representation. The output of this thesis is an implementation of the new memory representation which minimizes the complexity of the original solution.

Contents

1 Introduction 1

1.1 PHP 1

1.2 Static analysis 2

1.3 Weverca analyzer 2

1.4 Problem statement 3

1.5 Goals of thesis 4

2 Analysis of the PHP 5

2.1 Specialties of PHP 5

2.2 Mechanism of static analysis 8

2.3 Weverca analyzer 10

2.3.1 Architecture of weverca analyzer 10

2.3.2 Weverca worklist algorithm 13

3 Memory model for the Weverca analyzer 16

3.1 Memory location 16

3.2 Memory snapshot 17

3.2.1 Snapshot behavior 18

3.2.2 Structural data 20

3.2.3 Value storage 32

3.2.4 Functions and classes 33

3.2.5 Improvements of memory snapshot 34

3.3 Memory model algorithms 40

3.3.1 Memory path 40

3.3.2 Read 45

3.3.3 Assign 46

3.3.4 Extend 52

3.3.5 Merge *54*

3.3.6 Commit *64*

4 New memory model **66**

4.1 Modular implementation 66

4.1.1 Structure of modular copy memory model *67*

4.1.2 Basic interface *69*

4.1.3 Memory storage *72*

4.1.4 Memory algorithms *76*

4.2 Variants of implementation 79

4.3 Starting the analysis 84

4.3.1 Console application *84*

4.3.2 GUI application *85*

4.3.3 Web application *87*

5 Evaluation of memory model algorithms **89**

5.1 Heap sort 90

5.2 Travelling salesperson 92

5.3 School web page 94

5.4 NOCC webmail client 96

6 Conclusion **98**

Bibliography **100**

A. List of Images **101**

B. List of Tables **103**

C. Content of attached CD **104**

1 Introduction

As time comes the way of being a programmer evolved. Now a technology is everywhere and people are building more and more complex systems. Huge mistakes of the past and dozens of nameless examples of software issues cost so much money and human power that new programmers are trained to use techniques of software engineering. A good analysis of the problem, a correct documentation, a clean and a simple code can prevent creation of an error at the first place and speed the development up.

One way to minimize the number of errors is to use automatic tools which are able to go thru the whole code. Integrated development environments are not just text editors but complex systems which are able to provide real time scan of the source code. A highlighting of syntax errors has become a commonplace. Modern environments can go much deeper. Looking for uninitialized variables or a dead code shouldn't surprise any of us. An error pattern spotting is a powerful functionality which is able to save time of the modern programmer.

1.1 PHP

The PHP is still one of the most common web languages of these days. Together with the HTML, the CSS and the JavaScript forms an ecosystem for building complex web sites. There are lots of PHP programmers over the market and it is often a starting language for new programmers followed from its spread and a simple usability. All of these factors lead to a situation that the PHP is often default or even the only option for a new web project even that there are maybe better solutions based on modern OOP languages.

One of the huge disadvantages of the PHP comes from its great popularity. Huge widespread leads to the situation when many people use its weaknesses to break into the systems. These problems are well known and defense against them is quite simple. But even that there are lots of web pages which can be easily compromised because of its programmer didn't know or care about the security holes. Especially if a new programmer tries to build his own firs site there may be lots of security or logic problems within a code.

An automatic error spotting may be an answer for a current state. A tool which would be able to scan a whole PHP site and report potential security risks might have been useful for

all PHP programmers. But even that the PHP is so common language there is no tool which is able to do complex scan of the code.

1.2 Static analysis

A static analysis is a form of an analysis of a source code without the need of its executing. An analysis of running programs is called a dynamic analysis.

The main advantage of a static analysis is ability to determine some properties of the program from its source code. This ability is used in compilers to provide a code optimization or in integrated development environments in real-time analysis to produce complex warnings (a dead code, unused variables and more).

A static analysis of the PHP is special challenge especially because of the PHP is full of dynamic constructs. A weak typing and dynamic declarations are just beginning. A static analyzer of the PHP has to be very complex to handle all special cases and a memory representation has to be able to store different kind of information for each variable.

Despite this a static analysis can be used to spotting common security and runtime errors. A tool strong enough to spot major security risks would be extremely useful for any web developer.

1.3 Weverca analyzer

The Weverca analyzer (*Web verification tool for PHP*) was created in 2014 as a student software project in the *Faculty of Mathematics and Physics of Charles University in Prague* [4].

The Weverca is able to perform static analysis of a source code of a full web page written in the PHP and report possible security risks and runtime errors. The main advantages of the Weverca are:

- An ability to analyze a full page from a single source file
- The first PHP analyzer with full support of a PHP objects
- A modular architecture
- A framework for custom analyses
- A connection with phalanger compiler
- An open source code in the C# compilable in the Mono tool

- A plug-in for integration with the eclipse platform
- A possibility of a future grow – online checkers, an integration with development environments, new more complex analyses

The development of the Weverca analyzer still continues at the MFF UK in the *Department of Distributed and Dependable Systems*.

1.4 Problem statement

A m*emory model* is a part of a static analyzer which is responsible for storing values counted by an analysis. It has to be able to store all data which may be connected with an analyzed source code – possible values of variables, declarations, references and more. Special requirements for memory model of the Weverca analyzer follow the character of the PHP and a modularity of the Weverca analyzer.

The first version of Weverca analyzer contained two distinct memory models – the *Virtual reference memory model* and the *Copy memory model*. These implementations illustrated modularity of the Weverca tool and provided a basic functionality for the first release.

The virtual reference memory model is a minimal, the most simple implementation to illustrate possibilities of the Weverca analyzer. The copy memory model brings a more complex solution – copy semantics of references allows an analysis to achieve better results than using virtual references.

The First release of the Weverca was able to process simple examples of a PHP code. An improvements of the analysis brought possibility of analyzing a more complex sources. But at the end the limit of the memory model was reached. A big memory requirements and a huge time consumption blocked usability and further development of the Weverca analyzer.

The main problems of the copy memory model lie in:

- A huge memory grow
- A time complexity of algorithms
- Mixed data containers and algorithms
- A missing advanced optimizations
- Program errors
- Missing inner namespace hierarchy

1.5 Goals of thesis

All problems of the memory model have to be solved before the Weverca will be able to process real PHP sites. The situation is complicated by character of the copy memory model. Especially a too tight connection between an inner data representation and algorithms makes any optimization extremely hard. A new memory model was introduced to solve this limitation.

The new memory model follows a main philosophy of the old copy memory model and fixes its main problems which bring better possibilities for a future development.

The requirements on the new memory model include:

- Separate a structural data, a value storage and memory algorithms
- Refractor a namespace and a class hierarchy and fix program errors
- Allow to easily change parts of the memory model without affecting the others
- Identify time and memory bottle-necks and optimize containers and algorithms
- Implement a missing functionality of Weverca memory model
- Support advanced approaches of a static analysis to speed the analysis up
- Measure and compare a performance of old and new algorithms

This thesis will explain internal parts of a memory model, disadvantages of old approaches, possible solutions and comparison between them.

2 Analysis of the PHP

The Goal of the Weverca project is a creation of the first analyzer of the PHP language which would be strong enough to perform analysis of full PHP programs. The analyzer should be able to process all program statements, evaluate all possible values of all variables and highlight the constructs which may lead to unexpected behavior of a PHP program.

The main motivation to create such an analyzer is given by character of a typical PHP application. The PHP is mostly used to build the server side of a web application which is usually a gateway to some database accessed by application logic. The server side uses request parameters to perform a query to inner data storage. Note that some queries might be performed by any visitor but others require certain level of authorization.

A problem is that these applications are usually nonstop accessible with their public domain address. Anyone is able to try various input parameters in order to find how to compromise security of data. Most of security vulnerabilities are well known and can be avoided by several sanitization techniques. The most famous threads are: *SQL injection*, *Cross site scripting* and *Session hijacking* [2]. All of these threats try to pass a special string as a request parameter in hope that the authors of the web page haven't known about such problems.

Another use case of a PHP analyzer is to warn a programmer that there can be a runtime error caused by a propagation of a wrong value. Any OOP programmer surly experienced a null pointer, an index out of range or similar runtime exceptions. A static analysis is able to find that there is such possibility and warn the programmer in advance.

A motivation is also that a static analysis of PHP programs is a special challenge. The PHP programs are full of dynamic constructs which makes an analysis extremely hard. Some of these features are explained in the following chapter.

2.1 Specialties of PHP

At the beginning the PHP was a simple scripting language. The first use case was to add more options for a creator of a web page to show a dynamic content processed by the server. As time comes more and more features were added into the PHP. A current

state is that the PHP language contains all common constructs from modern languages and combines them with the principles of scripting languages.

All these constructs have to be used with care. The main danger of these methods is that an unskilled programmer may think that using them is a good idea and all programs should do it all the time.

The dynamic constructs are also very hard for a static analysis. A complexity of an analyzer increases with the number of dynamic constructs supported by a programming language. A challenging part is to analyze constructs which are not resolved until runtime. Some specialties of the PHP language which has to be considered include:

Weakly typed language

This is the main difference between compiled C-like languages and the PHP. A strong typing can prevent many errors just because a programmer has to declare the types of variables and is not allowed to use them to store values of another type.

The PHP has no static types and variables don't need to be declared at all. A programmer simply starts to use one to store anything. When used in expressions, the system casts the value of a variable by the current context. Cast priorities and results are specified within the documentation of the language.

For example these values are (sometimes) equal in PHP:

- Boolean false
- Empty string
- Number: 0
- String: "0"
- Null value

The main disadvantage of this behavior is that program may behave in many different ways than expected. A common problem is that a valid zero value may not pass a simple test that input value is not empty.

Object oriented programming

Objects and classes were introduced to the PHP in version 5. Since then it becomes a common way to improve the structure of the code. The PHP contains all main principles

and objects should not surprise a skilled OOP programmer. Note that all objects use the Java-like reference semantics.

Some confusion might be the combination of the OOP and a weak typing. A polymorphism in the PHP is not achieved by interfaces or an inheritance. Without a type check a programmer may simply declare classes with the same methods and a different code. A PHP interpreter selects the method based on the value stored in the variable.

Another difference is existence of an implicit object. Implicit object is created by the interpreter when assigning into the field of an empty variable.

An analysis has to handle cases that some calls may invoke different methods based on the class of the accessed object. Cycle iterating objects derived from the same base class and calling its method is a very common pattern.

Copy and reference semantics

Modern OOP languages (Java, C#) use the same approach to handle arrays and objects. An array is represented by a special object whose reference may be stored in a variable and used anywhere. The reference semantics of arrays is very convenient for a programmer when arrays are passed into or out of a function.

The PHP arrays are implemented with copy semantics similar to C++. The programmer has to use references (aliases) to pass arrays into functions otherwise a deep copy of the array is created. On the other hand, objects in the PHP use the reference semantics well known from OOP languages.

Includes and conditional declaration

The PHP allows splitting a code into several source files. Each file can contain a definition of a class or be a library of functions. The whole code is composed together by including all necessary files.

A surprising fact is that includes can be called with a variable file name or in different branches of a condition. Usage of this behavior is simple: programmer may create more libraries of the same set of functions and include the one which is needed. This was a way to bring OOP functionality to the old PHP without objects.

Conditional declarations should be used with special care because this may lead to the unexpected behavior of the program. The main danger is because of its simplicity.

Dynamic variables and calls

All modern languages contain a way to invoke functions dynamically. The Java and the C# contain a mechanism to access type information at runtime which allows the programmer to modify the internal state of an object. This access is quite slow and not recommended to use it if it is not necessary. Dynamic calls of the PHP are super simple. Scripting basics of the PHP allows the interpreter to resolve any name at the runtime. The programmer may use a variable instead of a function or a class name which means that the interpreter calls a function or class by the specified value.

For an analysis it can be tricky if a variable name can be resolved to multiple values. In that case the analysis has to read or update all matching variables or call all matching functions.

2.2 Mechanism of static analysis

A static analysis is a universal mechanism which is able to determine a custom property of analyzed source code. For example an analysis tool is able to resolve all possible values in all variables, to track data flow in program statements and more. A security analysis of imperative language can be performed by *Data Flow Analysis* and *Worklist Algorithm*, as described in *Principles of Program Analysis* [1].

Note that this text describes only basic techniques of a static analysis which are necessary to fully understand all requirements to a memory model We refer the reader to *Principles of Program Analysis* [1] to find more information about an analysis.

Data Flow Analysis is one of the four main approaches of static analysis described in [1]. An analyzed program is represented as an oriented graph called *Program Point Graph* (PPG). A node in PPG contains one statement. An edge describes program flow between statements. PPG is very similar to a control flow graph. The main difference is that any node in the control flow graph should contain several statements which belong to the same program block. PPG breaks program blocks to several nodes which allows iterating over statements one by one in *Worklist Algorithm*.

Worklist Algorithm is a universal algorithm which receives a finite constraint system S and produces total function ψ. The constraint system contains finite number of an inequations in form $x_i \sqsupseteq t_i$. An inequation consists of flow variable x_i and term t_i. Each t_i depends on a subset of flow variables. Produced function ψ maps each flow variable to some value of a complete lattice $(L, \sqsubseteq)$ with the least element $\bot$.

At the beginning, the algorithm selects ψ which maps all flow variables to the least element $\perp$ and puts all constraints to a worklist queue W. An iteration of the algorithm removes one constraint $x_i \sqsupseteq t_i$ from W and changes the value of $\psi(x_i)$ using the current ψ and term t_i. If the new value of $\psi(x_i)$ is different from the original one, all constraints influenced by x_i are inserted into W. The algorithm keeps iterating until W is empty which means that ψ has stopped changing and a fixpoint has been reached. Complete pseudo code of the abstract worklist algorithm can be found in [1].

The Worklist algorithm can be successfully used to process PPG of any imperative language. There are several ways to transfer PPG to a constraint system but the basic approach is very simple – flow variables represent nodes of PPG, terms represent both edges and statements.

Possible values of analyzed properties are given by the lattice. Each element of the lattice contains all information known in some node of PPG – for example state of the whole memory. The least element $\perp$ contains initial property value at the beginning of the analysis (*all variables with an initial or an unknown value*). If two elements of the lattice are comparable then **the larger** element contains **less** precise information (v*ariable values: {1}* $\sqsubseteq$ *{1,2}* $\sqsubseteq$ *INT* $\sqsubseteq$ *ANY*). The greatest element $\top$ represents a special *ANY* value which means that analysis is not able to resolve the value of the property (*all variables can contain anything*).

Processing PPG by worklist algorithm is shown in following pseudo code.

```
1.  worklistAlgorithm(PPG) {
2.    var worklist = [1, ..., nodes(PPG)];
3.    var values = [⊥| 1, ..., nodes(PPG)];
4.    while (worklist != []) {
5.      var node = extract(worklist);
6.      var newValue = evaluate(node, values, PPG);
7.      if (newValue != values[node]) {
8.        insert(flowChildren(node, PPG), worklist)
9.        values[node] = newValue;
10.     }
11.   }
12.   return values;
13. }
```

The Worklist algorithm uses the following functions:

- Function n*odes* returns number of nodes in given PPG.
- Functions flowChildren returns set of direct descendants of given node
- Function e*xtract* removes and returns one value from a given array
- Function *evaluate* computes out value of given node. Evaluation process depends on node statement and input edges of processed node.
- Function *insert* adds all given elements into given array

The big danger of this approach is an exponential grow of the memory and an infinite cycling of the fixpoint algorithm if the *evaluate* method keeps changing the set of found values. The *Evaluation* method should use special optimizations to reduce the amount of data and the number of changes. A disadvantage is that these optimizations usually reduce precision of a static analysis.

The analyzer has to carefully balance the level of grouping to be able to spot interesting patterns but still finish in a reasonable time.

2.3 Weverca analyzer

The Weverca analyzer is a framework for static analysis of a PHP program. It accepts a source code of whole PHP page and provides a static analysis to evaluate all possible values of all variables. It also contains support for custom second phase analysis (e.g. *Taint analysis*). The second phase is able to use all data from previous phase to find additional information about the code.

An output of an analysis is the set of analysis warnings and a program point graph representing the given source code. Each *program point* (node in PPG) contains two memory snapshots with a state of memory before and after the program point. The second phase returns the same PPG but all snapshots may contain additional data computed in the second phase (usually flags for variables and their values). The second phase is supposed to generate warnings related to the topic of an analysis.

2.3.1 Architecture of weverca analyzer

The Weverca analyzer consists of four main components: *Control flow graph builder*, *First phase analysis framework, Second phase analysis framework* and *Memory model.* A special part of the Weverca is the *Metrics* component which can be used as a standalone application to evaluate some parameters of a PHP program.

Control flow graph builder

The *Control flow graph builder* accepts a PHP code (as a string or an input file) and produces a control flow graph (CFG) with all basic blocks as nodes and flow directions as edges. Internally, the builder uses a part of the *Phalanger* compiler [5] to obtain an AST representation of a given source. The PHP code in the AST form can be easily transformed into a CFG and passed to the analysis framework or the metric component.

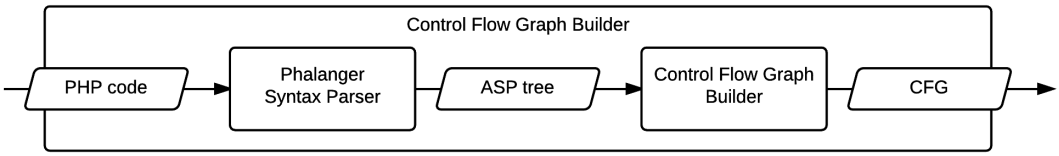


Image 2:1 Architecture of the Control flow graph builder

First Phase analysis framework

The *First Phase analysis framework* is a set of abstract functionalities with a single entry method. The analysis framework requires a control flow graph and a factory object for a Memory model. The result of the first phase analysis is a *program point graph* with resolved snapshots of possible values for each program point.

The first phase framework contains two components: a *Program point graph builder* and a *Fixpoint computer*. The fixpoint computer contains an implementation of the worklist algorithm which iterates over the PPG until a fixpoint is reached. A specific implementation of an analysis has to provide several resolvers which are called from the fixpoint computer. Each implementation has to provide an *Expression resolver* (evaluation of arithmetic statements, variables …), a *Function resolver* (resolving functions methods and providing calls), *Flow resolver* (conditions, exceptions, file includes) and a *Memory assistant* (providing custom services to a memory model).

The Weverca analyzer implements the first phase analysis in component called the *Forward analysis*.

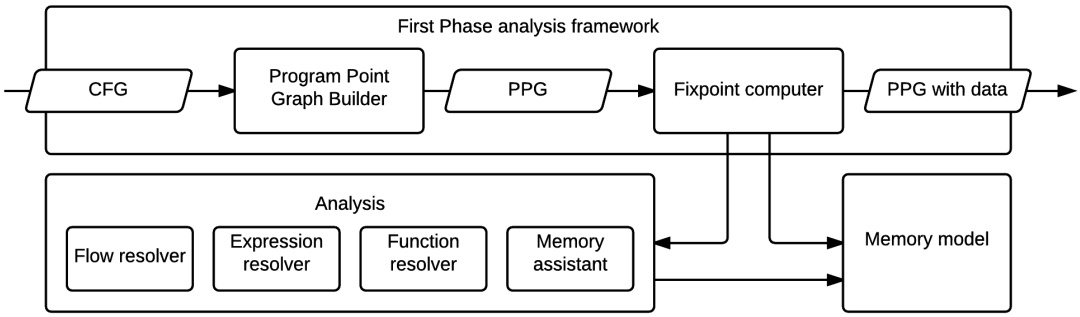


Image 2:2 An architecture of the first phase analysis framework

Second phase analysis framework

The *Second phase analysis framework* accepts a PPG with possible values from the previous phase. The result is a set of warnings and a possible modification of all snapshots to contain info values. The second phase framework contains another fixpoint computer which computes info values. All values from the previous phase cannot be modified but an analysis may use them to resolve which indexes or variables should be changed.

Weverca analyzer already contains an example implementation of the *Taint analysis*.

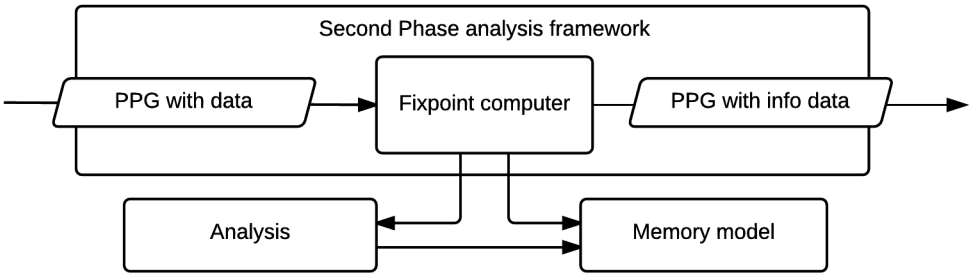


Image 2:3 An architecture of the second phase analysis framework

Memory model

The Memory model is a part of weverca which is responsible for storing counted values by worklist algorithm. Each instance of memory model is called *Snapshot* and contains all possible states of PHP memory at some program point. A snapshot keeps all values given by program statements such as used variables, arrays, objects, possible values and declarations.

The memory model has to satisfy public interface defined in the abstract base implementation of the snapshot class. The fixpoint algorithm and analysis resolvers use the public interface anytime if they need to read or update data stored in the snapshot.

The worklist algorithm also uses the public interface to perform flow operations – merging two snapshots into one, copying its content or managing memory transactions.

Internal parts of the memory model and all major operations are fully described in the chapter 3 Memory model for the Weverca analyzer.

2.3.2 Weverca worklist algorithm

Both phases of analysis contain an implementation of the worklist algorithm to find a fixpoint of memory models. The Worklist algorithm uses a similar construct as described in chapter (2.2 Mechanism of static analysis). But there are certain differences which need further explanation.

A program point graph produced by a builder component consists of a collection of program point objects. Each program point represents a single program statement or a start or an end of some program branch. A program point graph has knowledge about program points at the beginning and the end of an analyzed source code.

The analysis framework contains definition of an abstract program point and several distinct implementations to represent various PHP statements. Each implementation holds any data which are necessary to perform an evaluate method of a worklist algorithm. The abstract program requires that each program point has to contain *in* and *out* snapshots and collections of *flow children* and *flow parents*.

An in-snapshot contains whole memory state at the beginning of a program point. An out-snapshot represents memory state after evaluation of a statement. Flow parents are all program points from which leads an edge to the current program point. Flow children are all direct descendants of the current node.

An in-snapshot has to be always constructed before a program point statement is evaluated. At the beginning of a worklist algorithm a new initial snapshot instance is created and stored into the input snapshot within the first program point of the PPG. The Input snapshots of all other program points are given by out snapshots of their flow parents. If a program point has only one parent then the input is just an extension (copy) of the parent's out-snapshot. When there are multiple parents then analysis performs a merge operation to get a single input snapshot given by the combination of all outputs.

The following code shows a simplified worklist algorithm as it is implemented within the Weverca analyzer. Note that the algorithm calls a public interface of a memory model to invoke memory operations: *startTransaction*, *extend* and *commitTransaction*. The Extend method invokes the *extend* or *merge* memory operation depending on the number of given snapshots. Other memory operations might be called when a program point is evaluated within its method *flowTrough*. The memory operations are described in chapter 3.3 Memory model algorithms.

```
1.  worklistAlgorithm(PPG) {
2.    var worklist = [PPG.Start];
3.    while (worklist != []) {
4.      var programPoint = extract(worklist);

        // Prepares input snapshot of given program point
5.      var parentOutputs = getOutputs(programPoint.FlowParents);
6.      programPoint.InSnapshot.StartTransaction();
7.      programPoint.InSnapshot.Extend(parentOutputs);
8.      programPoint.InSnapshot.CommitTransaction();

        // Flows thru program point and uses input to evaluate output
9.      programPoint.OutSnapshot.StartTransaction();
10.     programPoint.FlowThrough();

        //Commits transaction and if output differs then enqueue all children
11.     var outputChanged = programPoint.OutSnapshot.CommitTransaction();
12.     if (outputChanged) {
13.       insert(programPoint.FlowChildren, worklist);
14.     }
15.   }
16. }
```

The implementation uses the same helper functions *extract* and *insert* to operate with a worklist queue. Function *getOutputs* gets an array of output snapshots from a given collection of program points.

Another difference between the current and previous worklist implementation is the initial content of the worklist queue. The previous implementation inserted all program points to the queue but weverca uses only the starting one. Both approaches give valid results but there are certain differences which have to be considered.

The abstract worklist algorithm has no knowledge about a program flow and an ordering of program points. The points can be processed in any order – usually given by LIFO or FIFO implementation of a worklist queue. The algorithm guaranties that all nodes will be processed and a computation will continue until the fixpoint is reached. An disadvantage of this approach is that the algorithm may try to evaluate some point before its flow parents. This won't affect the computation, because the point is processed again when the parents are ready. A problem is that it may slow down the computation when this happens very often.

The Weverca processes program points by order given by their position within the analyzed code. It also tries to use an intelligent re-ordering of a worklist queue to avoid unnecessary processing of program points. The Weverca approach also skips inaccessible program points – program points with no parents. Skipped program points might be reported to the user as dead code warnings.

3 Memory model for the Weverca analyzer

A process of a static analysis produces huge amount of data – declared types and variables, possible values and other special data computed by the analysis. This data has to be stored to allow the analysis to access any information when it is needed.

Moving memory functionality to a memory model brings possibility that the analysis may remain as simple as possible and focus to a program flow. Any algorithm connected with a memory has to be part of the memory model which is responsible for keeping the memory in a consistent state.

This chapter will describe principles of the memory model which may be used to store state of a PHP program. Basic principles are based on the original Copy memory model.

3.1 Memory location

A memory location is an abstraction of a single memory entry. It is a basic element of the memory model. It is used to identify and store any data connected with each variable, a cell of an array or a field of an object.

Identification of memory location

Correct functionality of the memory model requires that memory locations have to be defined deterministically. The deterministic character of memory locations is necessary to allow the memory model to be able to easily handle locations and relationships which was created in different branches of code – especially for languages which allow using variables without declarations (which is the case of the PHP).

Each distinct implementation of the memory model has its way to ensure deterministic behavior. Main difference between the Virtual reference and the copy memory models is the process to distinguish between memory locations.

The simplest approach is to use memory locations based on a name of variable, an index of an array or a field of an object. The copy memory model uses PHP access paths to define each memory location. If two memory locations are defined by the **same access path** then is the **same memory location**. This approach follows a fundamental principle of the copy memory model which is strongly focused for ensuring read-write semantics (see chapter 3.2.1 Snapshot behavior).

PHP access path

A PHP access path is a sequence of identifiers which leads to a memory location.

1. A variable usage is the simplest example of an access path
    - $*variableName*
2. Any access path may be extended by an array access
    - *AccessPath* [*index*]
3. Any access path may be extended by a field access
    - *AccessPath->fieldname*
4. A variable name, an index or field may be replaced by an access path

The access paths without nested paths (which follow rule 1, 2 and 3) are called *Simple access paths*.

The code below shows some examples of valid access paths:

```
17. $variable
18. $variable['index’]
19. $variable['level1']['level2']
20. $variable[$variable2]
21. $variable[$variable2['index']]
22. $variable->field
23. $variable->field[‘index’]
24. $variable->field1[‘index’]->field2
25. $variable->$field
```

Access paths are common part of a PHP source code which uses them in program statements to select a memory location to be read or updated.

3.2 Memory snapshot

A basic unit of the memory model is called *Memory snapshot*. Any snapshot contains whole state of memory in some program point. An analysis interacts with snapshots and modifies inner data by statements presented in program points.

At the end an analysis contains two distinct snapshots for each program point – its input state and output state. These data may be presented to a programmer or used as a source for the next phase.

The memory snapshot has to be able to store all information necessary for an analysis. List of information may change with type of an analysis and an analyzed language.

3.2.1 Snapshot behavior

An output of the Weverca analyzer is the construction of a program point graph with associated snapshots for each program point. Computed data describe possible behavior of a PHP program. Next step may be presenting data to a user or a custom analysis over computed snapshots.

The Weverca analyzer process program points one by one by the possible flow of the program. Each program point contains two distinct snapshots – an input and an output memory state. An input state is mirrored output of the previous snapshot or product of merge of several output states. Output state is modified input state by a program point statement.

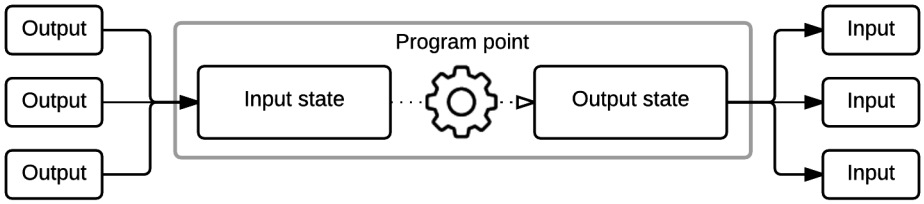


Image 3:1 An input and an output state of a program point

As analysis continues the analyzer has to be able to access any relevant snapshot to get the data and apply changes to the modified one. A critical requirement for the snapshot implementation is: *Modification of one snapshot cannot change any other*. A deep copy of inner structures from one snapshot to another is the simplest way to fulfill this rule.

Transactions

Transactions are used to allow analysis to permit modifications of the single snapshot at time. Any update has to be performed within an opened transaction. The analysis is still able to read any value after committing the transaction but for updating has to start the new one.

The second usage of transactions is to group update operations together and to check for changes within the transaction. This is critical for the worklist algorithm. It needs to know whether some statement changed the state of a memory due to the state in previous

iterations of the worklist algorithm. The commit phase of the transaction is the right place where to determine whether the state of the snapshot is the same as before.

Write read semantics

A write-read semantics means that when a program writes a value to some memory location then it should be able to read the same value back. An analysis may have problem to ensure the write-read semantics of a memory location when some code wants to write to an uncertain memory locations.

Following code shows an example of corrupted write read semantics.

```
1. $a = 0; $b = 0; $c = 0;
2. if ($_POST[?]) $a = &$b;
3. else $a = &$c;
4. $a = 1;
```

The program contains two *MAY* aliases between variables. At the end there will be two possible values for variables B and C but only one possible value for variable A.

The virtual reference memory model has a *weak write-read semantics*. Creation of a new alias creates a new *virtual reference* between aliased memory locations. At the end of the analysis there will be two weak connections from the variable A to other variables. When the analysis tries to assign a next value it is not able to determine which place should be strongly updated so it has to update both memory locations. As a result the analysis will have information that the variable A may contain both values.

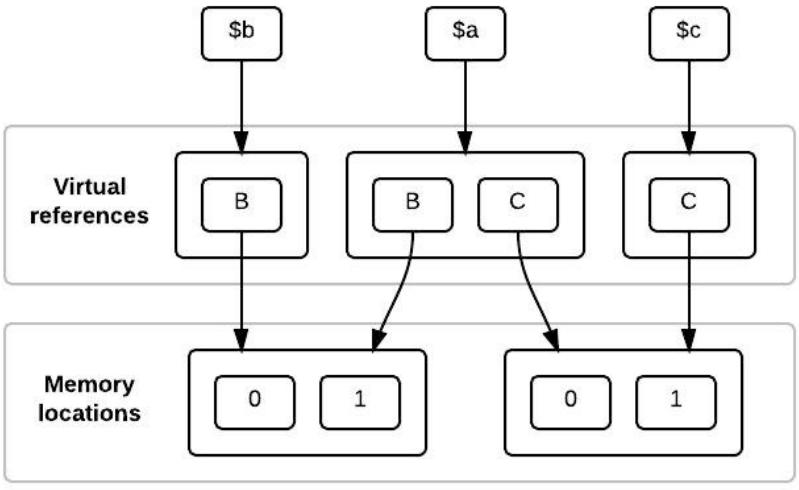


Image 3:2 A memory representation of the Virtual References memory model

An original design of the copy memory model was meant to ensure the write-read semantics of each memory location. To ensure the write-read semantics the memory model distributes every possible value across all connected locations. Side effect of distributing means that sometimes the same value is *copied* to several memory locations. Ensuring the write-read behavior increases precision of the analysis but consumes more memory and an update of a memory location is more time complex.

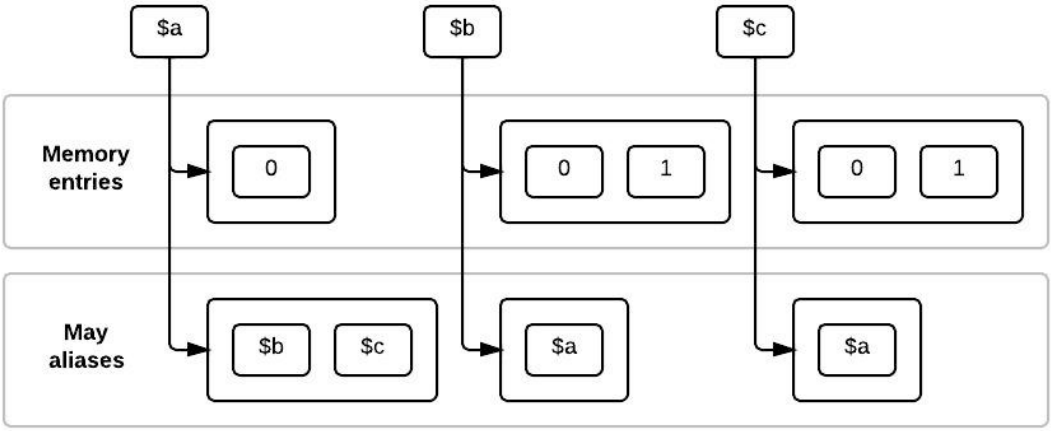


Image 3:3 A memory representation within the Copy memory model

The main invariant of the copy memory model says: *There is the unique memory location for each variable, index or field in the memory model.*

3.2.2 Structural data

Structural data are special values describing a characteristic of each memory location and its relations within the snapshot. A collection of structural data can be interpreted as an oriented graph where nodes are memory locations and edges are relations between them. The memory model has to be able to store all basic structural information as:

- A Name and a type of the memory locations (whether is a variable, a constant, an index of an array, a field of an object …)
- A definition of an array (a parent memory location and a list of indexes)
- A definition of an object (a class of the object, a parent memory location and a list of fields)
- References between memory locations (a list of referenced locations and information whether a reference is strong or weak)

3.2.2.1 Variables

Usage of variables in the PHP is similar to any other programming language. The main difference is that a variable doesn't need to be declared and any variable can contain a value of any type. In the PHP may easily happen that some program flows may lead to state that a variable may remain undefined.

```
1. if (?) { $a = 1; }
2. echo $a;                                    $a = { undefined, 1 }
```

The structure of the memory model has to contain a list of all defined variables and be able to map a variable name to an appropriate memory location. Each snapshot has an associative container with possible variable names and matching memory locations. The name mapping is critical for the read and update algorithms which need to resolve all matching variables for a given access path.

Unknown variable

An *Unknown variable* is special memory location which is used to store values of a variable which name cannot be resolved. The memory model uses all information connected to an unknown variable even when an undefined variable is accessed.

The unknown variable is common pattern of memory model. A memory representation of arrays and objects contains a special *unknown index* and *unknown field* with the same semantics. Each array and object variable is connected with its own *unknown memory location*. An access to an unknown memory location of other array object is permitted.

- An unknown location is always marked as a *undefined*
- Any update of an unknown location is *WEAK*
- Any update is distributed to all other variables within the same variable container
- An unknown variable is source of data for undefined locations in the same container
- A newly created location in the same container is filled with data from an unknown variable

There are more usages and limitations of unknown locations which will be described together with related memory algorithms (see chapter 3.3 Memory model algorithms).

Special variables

The collection of variables is used to store all data related to variables which may be used in a PHP code. The memory model contains several distinct collections to store different information computed by the analysis. These collections may be used by the analysis or the memory model to store any information connected with processed program point. A separation of this information resolves conflicts between variables created from a code or the analysis.

There are three types of variables in copy memory model:

- Variables
    - Identified by a variable name
    - Standard variables from a PHP source code
- Control variables
    - Identified by a variable name
    - Special variables from a source code
        - Static variables, constants
    - Flow information
        - Script name, function name, call depth
    - Another information which may be related to a program point
- Temporary variables
    - Identified by a generated ID
    - For internal usage of the memory model and its algorithms
    - Any information which has to be stored in inner structure but cannot be connected with any memory location

3.2.2.2 Arrays

Arrays are simplest structural type which is widely used by all programmers in various programming languages. An array is an associative collection which maps an index (a key) to some memory location (a value). The structural part of the memory model has to be able to store a definition of each array. The definition of the array contains all information necessary to allow good functionality of the analysis and memory model algorithms.

The memory model for the Weverca tool has some special requirements based by a dynamic character of the PHP.

Associative container

In the PHP a cell of an array can be added or removed anytime. Even any primitive value can be used as an index of an array. An indexing by any primitive type is quite straight forward and may be processed the similar as names of variables or fields of objects.

The dynamic character of array indexes has special requirements to the memory model. The structure of the memory model has to be able to change a structural description of an array after creation. The situation is complicated because any manipulation with the structure of the array mustn't affect previous instances of the snapshot. Another complication is driven by the possibility of creating an index in the branch of some condition. The memory model has to be able to merge the different descriptions of the same array.

To solve these problems each array in the Weverca consists of two parts – identification and a memory representation. The identification is simple instance created by the analysis when an array is declared. The analysis and the memory model uses created instances to identify each array anywhere in the inner structure or even in distinct snapshots. The memory model is able to map an identifier to a special description object (*descriptor*) with all related data. When a definition of an array is changed then the related snapshot will create new version of the descriptor object and change the mapping of an array. It is possible to use the identification object to access the array in any program state in that the array is defined.

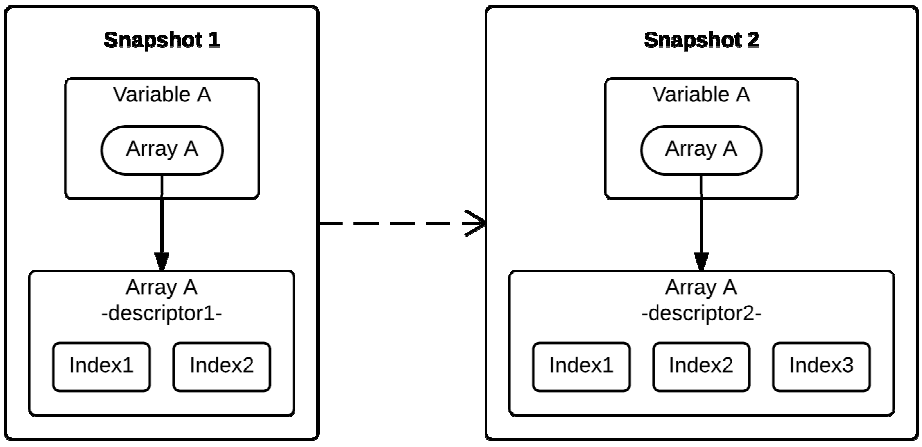


Image 3:4 An array identification and description

Copy semantic

As mentioned before the PHP uses mixed semantic for structural types – a copy semantic of arrays and a reference semantic of objects (see chapter 2.1 Specialties of PHP). The copy semantics of an array means that every array in the PHP is tightly connected with an enclosing memory location. The memory model has to support this requirement by creating a deep copy of an array when array is assigned between variables. Assigning of arrays is discussed in a description of an assign algorithm (see chapter 3.3.3 Assign).

Unbounded depth

The PHP is a weakly typed language which means that any memory location may contain a value of any type including an instance of an associative array. Even each index of an array may contain another array. The weak typing and an associative behavior of arrays bring possibility of creating a multi-dimensional array with a varying depth and mixed types. Before OOP was introduced the associative containers presented way how to store page data and share them across whole code. The memory model has to allow an unbounded depth of an array to satisfy a weak typed behavior of the PHP.

Conditional creation

In the program flow may happen that two possible arrays for a single memory location may be created in several distinct conditional branches.

```
3.  if ($condition) $a = array(1,2);
4.  else $a = array(3,4,5);
5.  echo $a[1];                                  $a = {array({1,3}, {2,4}, {-,5})}
```

There are several approaches which may be used when the memory model detects that memory location may contain distinct arrays.

The virtual reference memory model stores both arrays as possible values of a memory location. Each access to the memory location has to be prepared to the situation that there can be more than one array in the location.

Philosophy of the copy memory model says that each memory location contains every possible value. That is why the copy memory model merges all possible arrays into one.

In the copy memory model there always will be maximally one array for a memory location.

3.2.2.3 Objects

Objects represent another well known pattern of a structural type which is a common part of huge family OOP languages. Programmers started to use all advantages of OOP since objects were introduced in the PHP version 5. A curious fact is that the Weverca is the first analyzer of the PHP with full support of PHP objects.

Dynamic fields

Each object is internally represented as an associative array of fields. A field may be declared within the class or be added anywhere in the code. The dynamic character of object fields allows the memory model to handle each object similar as in case of associative arrays.

Each object consists of identification and a descriptor. The identification is created by the analysis when the object is created. The descriptor contains a list of all used fields and a type of the declared object. The snapshot maps the identification to the descriptor which allows adding new field to an existing object without a massive knocking effect across whole snapshot.

Reference semantic

A reference semantic is the main difference between objects and arrays from the memory model point of view. The copy semantics of an array means that the PHP always creates deep copy of the array when is assigned. Any change on the copied array won't be propagated to the old one. In case of an object the PHP just copies the reference which means that the object may be modified from both variables.

```
1. $arr = array();
2. $obj = new object();
3. $arr2 = $arr;
4. $obj2 = $obj;

5. $arr2[1] = 1;
6. $obj2->a = 1;
```

The copy semantic causes that ARR and ARR2 contain different arrays – an assigned value will be inserted only to the array in ARR2. OBJ and OBJ2 contain the same object because of reference semantics.

The first concept of the copy memory model was meant to use the same approach for identifying fields of objects as any other memory location – by their access path. Every

memory location was meant to contain maximally one object with merged information from every possible object. Any update to each memory location should have been distributed to locations with references to the same object. The original approach followed basic philosophy of the copy memory model and was meant to ensure the write-read semantic even for the PHP objects.

The approach of the first concept was not usable because of the typical usage of the PHP objects. Any object may be referenced from any place – even from another object. A structure made of object references is a common pattern in the OOP code.

Let's see what happens when a PHP code will try to iterate a linked list. The copy-like approach would have lead to copying sequence of objects when structure would have been traversed by a cursor variable. Each assign to the cursor variable would have lead to deleting an old content and copying next part of the structure. The copying might have been solved by optimization but the main problem is possibility of a cyclic reference in the linked list structure which may lead to an infinite depth of a memory path.

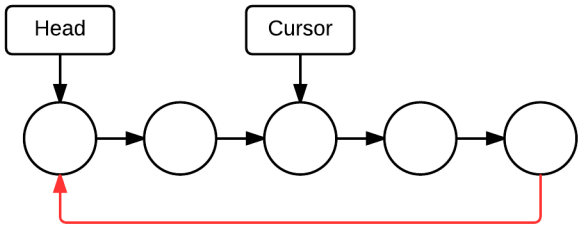


Image 3:5 A linked list with a cyclic reference

The simplest solution of these problems was to support only the weak write-read semantic for objects.

- Each memory location may contain any number of objects
- Distinct memory locations may contain the same object
- Objects which was created in different branches are always distinct objects
- Simple access path with field access may be resolved to several memory locations
- Location of any field is identified by object identification and field name
- If some memory location may contain multiple objects then a WEAK update has to be performed

3.2.2.4 Aliases

Aliases in the PHP are very similar to references in C++. The main use case is the same – to prevent copying of structural types when passed in or out from a function.

Arrays and read-write semantics

Aliases together with the weak typing bring possibility to change default copy semantics of arrays to the reference semantics.

```
1. $arr = array();
2. $arr2 = & $arr;
3. $arr2[1] = 1;
```

Forced reference semantics causes that ARR and ARR2 contains same array. Update on line 3 has to modify two memory locations.

The copy memory model still tries to ensure the strong write-read semantic for arrays and aliases even thought it may lead to all problems described in the previous chapter. Inner algorithms have to support cyclic references without stack in an infinite loop. This is in contradiction to objects approach however typical use case of arrays and aliases is not to build complex structures but to prevent copying the simple structure of a single array.

Any other usage is suspicious and should be replaced by objects with an implicit reference semantic.

- The copy memory model internally represents aliases as links between memory locations
- The read-write semantic is ensured by copying all possible values between connected locations
- A structure of an array is deeply copied to an aliased location
- Cyclic references are resolved by algorithms
- When a memory location is updated then all its aliases are updated too

Must and may aliases

There are two types of aliases – *MUST* and *MAY*. A *MUST* alias is alias which has to exist in current program flow. A *MAY* alias is the one which may not exist in some flows.

```
1. $mustAlias = & $variable;
2. if (?) { $mayAlias = & $variable; }

3. $variable = 1;
```

The difference between MUST and MAY aliases is illustrated on line 3 of an example code. At the end *variable* and *mustAlias* will contain value 1. Variable *mayAlias* will contain value 1 or stay undefined.

The structure of the memory model has to distinguish between *MUST* or *MAY* aliases. Algorithms have to support the aliasing by providing a strong or a weak version of a requested operation.

3.2.2.5 Memory stack

A memory representation of a typical OOP language consists of two parts – a *stack* and a *heap*. The stack usually stores primitive values and pointers to the heap which are used in the current local context. The heap contains all instances of structural types – objects and arrays. The local context is always deleted on the exit of the current block but the heap has to be handled manually (C++) or by a *garbage collector* (Java, C#). Some languages (C++) allow storing structural data even into the local context of stack.

Memory stack in PHP

The memory model uses a stack to separate *local contexts* of functions. Each function has its own level of the stack to store declared variables and arrays stored in local variables. Any memory location which has to be transferred into or out of the local context has to be deeply copied. Object instances are stored on heap – in or out of function are passed by reference.

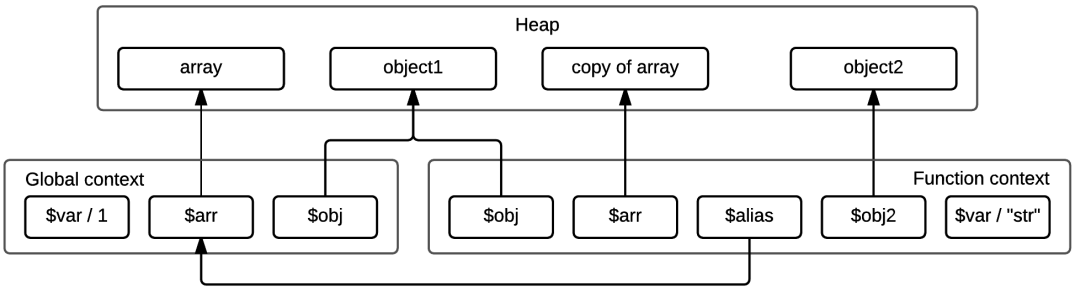


Image 3:6 Stack and heap in PHP

A communication between stack levels is normally forbidden however the PHP has a mechanism how to modify the local level of another function. Global variables or aliases are strong enough to pass references in or out to the function. A common use case of references in any language is to pass array into function without the need of deep copying.

The memory model has to allow updating or even creating a new array in a memory location related with another stack level.

```
1. function foo( &$a ) { $a[1] = 1; }
2. foo($v);
```

Code above illustrates adding index into an array passed by reference. There is possibility that the new array will be created in the calling stack level when the variable V will be undefined.

Identification of stack level

Each snapshot of the copy memory model consists of the *heap of objects* and list of the *stack levels*. Each level of the memory stack has its own collection of declared variables and arrays.

A correct functionality of the memory model requires that snapshot has to be able to find a stack level of a memory location. An identification of the stack level has to be added into each memory location of a variable or an array index. The simplest way how to identify each stack level is to count the number of function calls. Each snapshot holds a *call level* which is incremented on any function call and decremented on return. Each stack level will receive the call level of its enclosing snapshot. The identification of the stack level is used to allow the memory model to distinguish between variables with same names but stored on different stack levels.

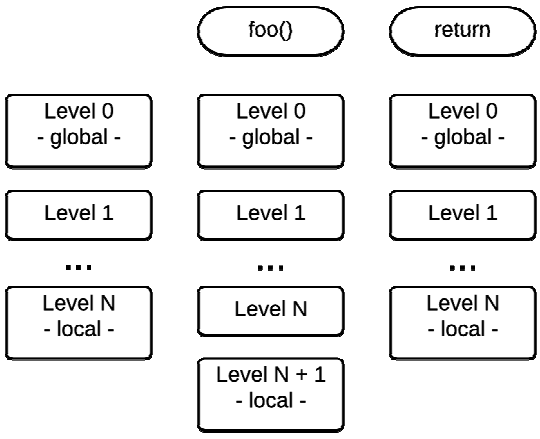


Image 3:7 Call levels in a call stack

Definition of call level

- An initial value of call level is 0 – a *global context*
- A call level of a snapshot is incremented when new function is entered
- A call level of a snapshot is decremented when function is returned
- A call level may never be less than a zero

Life cycle of stack level

- A snapshot has to contain a stack level for every value from 0 to the call level
- A stack level is identified by the call level of a snapshot when a function was entered
- A local stack context is identified by the current call level
- A new local level is created when function is entered
- A Local level is deleted when function is exited
    - All variables and arrays of this level has to be deleted
    - All alias connections has to be deleted

3.2.2.6 Memory tree

The final abstraction of memory state of single snapshot is called a *memory tree*. The memory tree is an oriented graph which describes relations between memory locations. Aliases and object references may cause that the memory tree contains relations between branches however typical processing of memory is from a root to leaves.

Following chapter describes parts of a memory tree.

Stack and heap

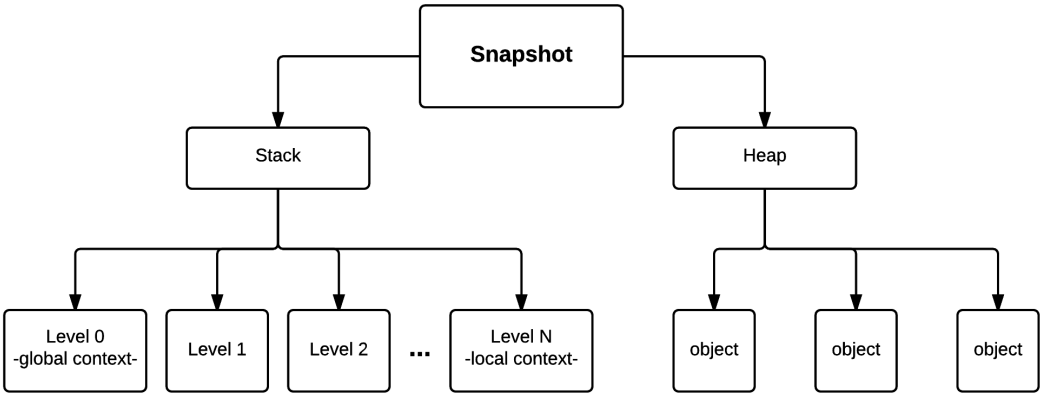


Image 3:8 Top level branches of a memory tree

There are two main branches in each memory tree – a stack and a heap. The stack branch contains sequence of stack levels from a global to a local context. The heap contains each object which is part of a memory snapshot.

Collection of locations

A collection of memory location is a list of named nodes with the same parent. Each node represents some memory location of the memory model. Each collection always contains special node for an unknown memory location.

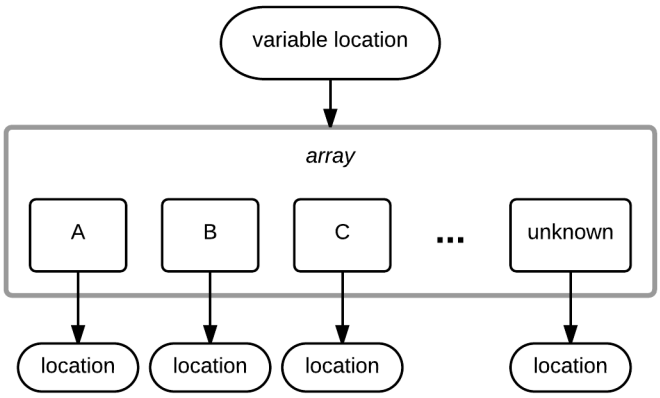


Image 3:9 An example of an array stored in a variable

Collection of locations is common pattern which may be spotted within memory tree – variables, arrays or objects.

Example of memory tree

Image 3:10 contains a simplified example of a memory tree. Green dashed lines represent object references. A red dotted line is an alias link between a variable and an index of an array.

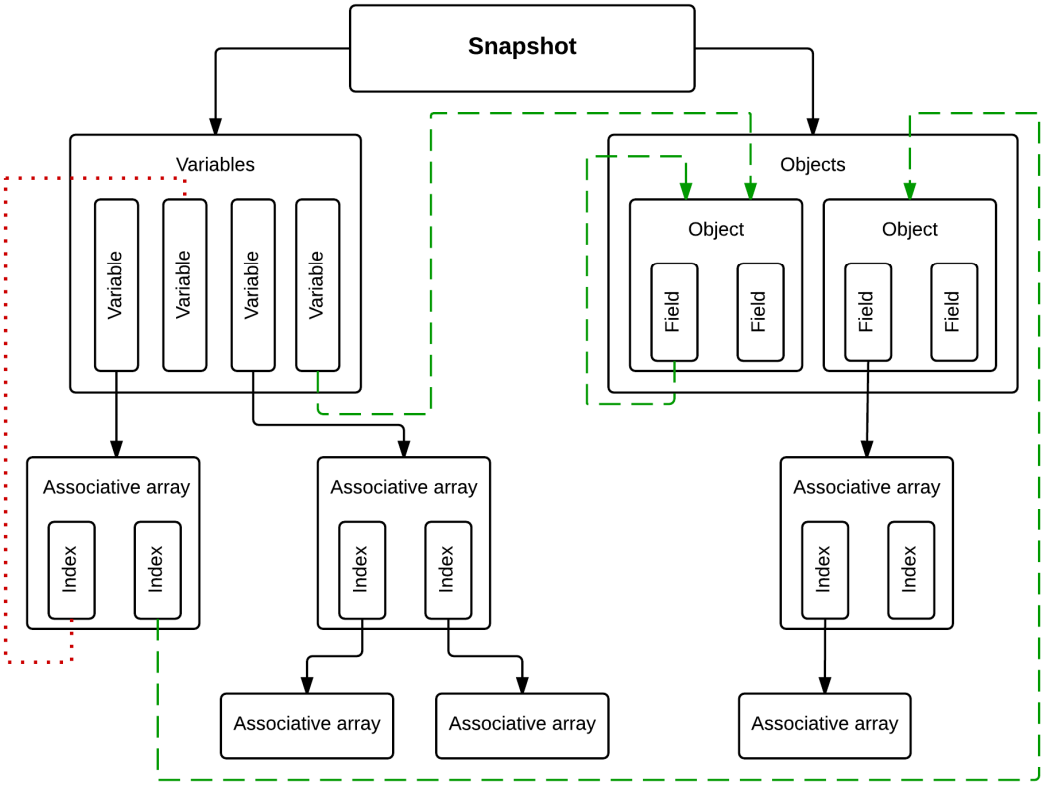


Image 3:10 An Example of simplified memory tree

3.2.3 Value storage

Each snapshot consists of two parts – structural data and value storage. The structural data defines possible memory locations and relations between them. The value storage maps known memory locations to possible values computed by the analysis.

The analysis reads or updates values of locations. At the end each location will contain a set of values which may appear in each memory location (a variable, an array or a field).

```
1.  $a = 1;                          $a = { 1 }
2.  if (?) { $a = 2; }               $a = { 2 }
3.  echo $a;                         $a = { 1, 2 }
```

The number of possible values has special meaning for the analysis. Based on number of values each memory location can be: *Undefined*, *Strong Value* or *Weak Value*. A location is undefined if is no value is stored within the location or if it contains only a special Undefined value. The analysis and memory algorithms have to use special approach to handle undefined locations.

A strong value means that a location contains only primitive values (number, string, boolean …), an array or a single object. If the location contains multiple values then all values are weak.

There are some special values which are always weak even if a location doesn't contain any other value. Weak values are: numeric interval, *any value*, *any string value* and other uncertain or aggregated values.

Second phase of analysis

The first phase of the analysis collects only values which may appear in the PHP source code. The second phase of analysis operates with its own set of values which brings additional information to locations. Special values of the second phase are *info flags* which are computed over the known structure and the set of possible values.

The memory model supports the second phase by introducing secondary info value storage. Behavior of info values is identical to the main value container. The analysis just needs to switch state of the snapshot to access the storage of the info values.

3.2.4 Functions and classes

The dynamicity of the PHP allows a programmer to declare new functions and types in conditional branches. This is another difference between PHP and compiled languages which don't have such ability. Templates and generic are powerful construct but they still need to be resolved at compilation time.

```
1. if (?) { function foo() { return "A"; } }
2. else { function foo() { return 1; } }
3. $v = foo();                                  $v = { 1, "A" }
```

The possibility of conditional declarations in the PHP means that the analysis has to resolve possible declarations for each function or class call. Declared functions and classes have to be part of the snapshot. An implementation may be really simple – an associative container which maps qualified names to lists of declarations.

3.2.5 Improvements of memory snapshot

The first release of the Weverca analyzer contained a basic implementation of the copy memory model which followed all basic principles described above. It was a minimal working implementation of the memory model and it wasn't ideal. Further development of the Weverca and attempts to analyze real PHP programs discovered performance and memory issues of the original memory model.

This chapter will describe advanced techniques which may be used to improve a performance of the memory model. All techniques are introduced from the snapshot point of view only. Any other performance impacts are discussed in chapter describing a related memory algorithm (see chapter 3.3 Memory model algorithms).

3.2.5.1 Data sharing

The whole memory model has to store massive amount of data. Each program point contains two snapshots with a complete state of a possible memory. Sharing unchanged data between snapshots represents obvious way how to reduce memory complexity of memory model snapshots.

Data sharing in original copy memory model

The original copy memory model tried to reduce the memory complexity by sharing immutable version of object and array descriptors and definitions of memory locations. Other components were copied from a source snapshot when an analysis requested a new copy of whole snapshot.

Any modification of shared structures had to provide a writeable copy of an original container. The writeable copy might have been changed, converted to an immutable version and stored back to the modified snapshot. Immutable containers fully satisfy the requirement for snapshot separation.

Based on further development was found that an original approach is not strong enough. A problem is that copied components of a snapshot represents significant share of data in the snapshot – mostly associative containers or high level objects.

Laziness

Laziness is an approach which tries to delay performing some operation until the result is needed. This approach may reduce a time and a memory complexity when a process

contains an operation which result may became deprecated before the first usage. The new memory model uses laziness to delay copying of inner structures until modification.

A lazy structure consists of two parts – a proxy and a data object. The data object is an original version with all inner data and defined read and write operations. The proxy object contains reference to the data object. Other parts of a system communicate only with the proxy. All operations are mirrored to the inner data object.

Making copy of the proxy means that new a proxy object contains a read-only reference to the data object from the source proxy. The read-only reference means that any write operation invokes a creation of a copy of an inner object which replaces the read-only reference.

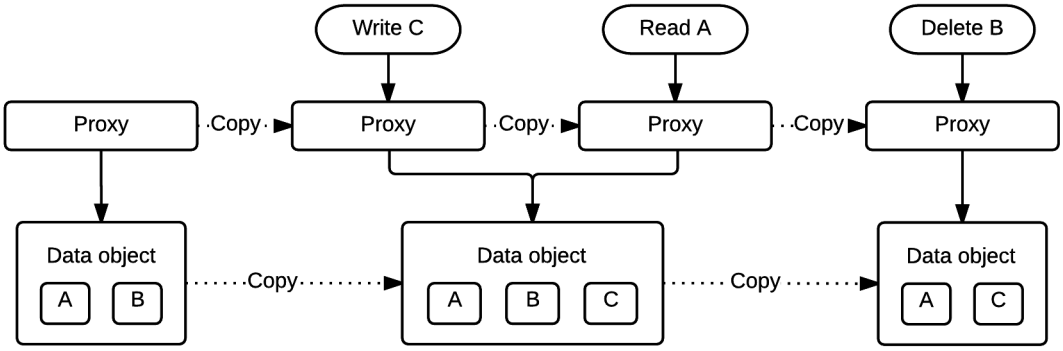


Image 3:11 Lazy structure

The full power of laziness is revealed when lazy objects are linked in a hierarchical structure. Laziness may cause that whole branch of a tree structure may not be copied but shared across many instances. The copy memory model may use laziness in all levels of a memory tree.

The new implementation of the memory model uses laziness for inner associative containers, sets or even high level enclosing objects. Adding laziness to the copy memory model had significant performance impact to the analysis (see chapter 5 Evaluation of memory model algorithms).

Partial sharing

Lazy structures are very useful when the number of write operations is low in comparison to the number of copying. When number of writes increases laziness loses its effect. Partial

sharing of structure is a technique which may be used to improve lazy structures to be more efficient.

A disadvantage of laziness is that a proxy still needs to provide the full copy of a data object on first write operation. Making copy of whole container may not be necessary when a write operation wants to access only the single inner object.

Next improvement of inner memory containers is to split an inner structure to buckets with lazy behavior. All inner objects are grouped by a hash function and distributed to buckets. A write operation will copy only the modified bucket. Unchanged buckets may be shared across many instances of the container.

The new memory model does not implement this behavior. There is possibility for next development to improve high level containers by usage of a partial sharing when will be necessary.

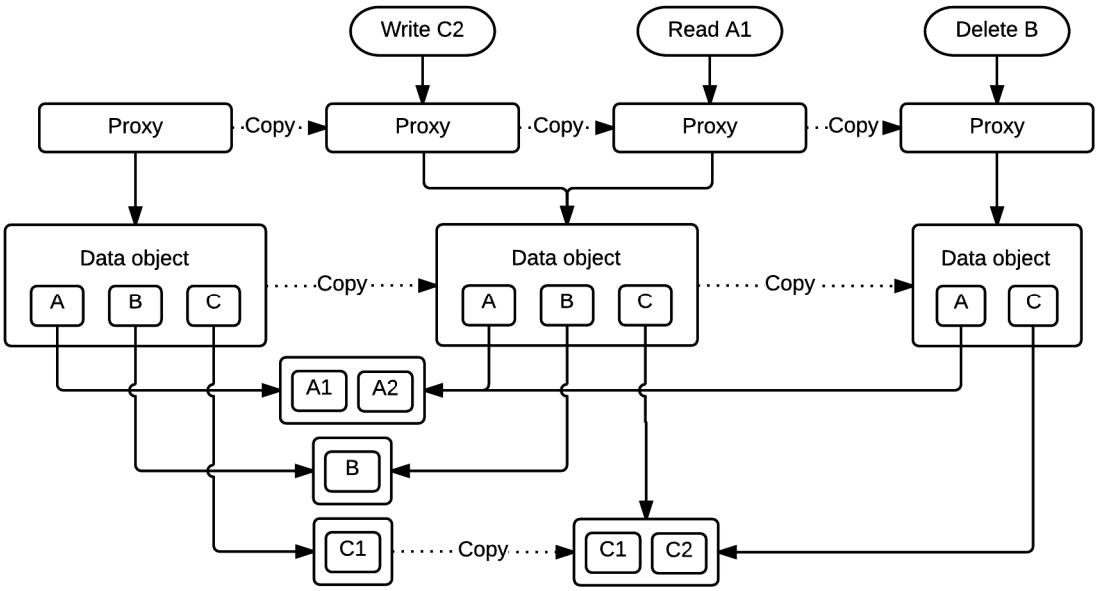


Image 3:12 Partial sharing

Differential containers

Differential containers are another way how to reduce the number of copying in lazy container. Each version of a differential container contains a link to its ancestor and a list of changes. Reading means to find proper container with the last version of the value or information that the value was deleted.

Complexity of a read may be reduced by using a limit for number of containers in a row. The data are merged to a single container when the length of the sequence exceeds the specified limit.

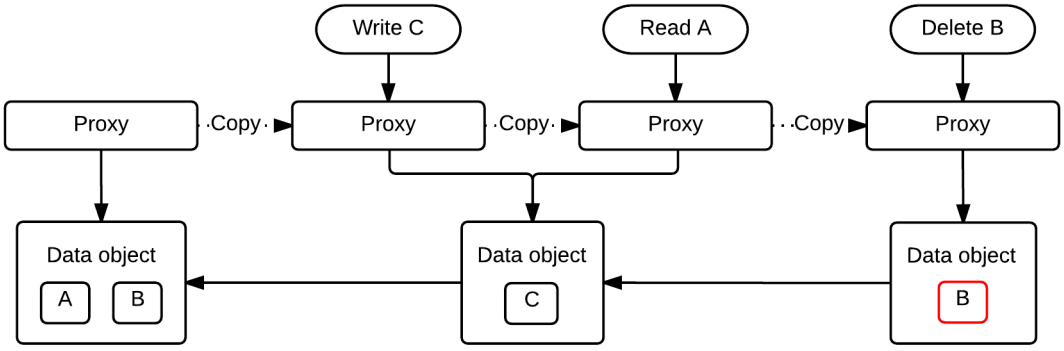


Image 3:13 Differential containers

3.2.5.2 Tracking changes

A tracking changes is an ability of the snapshot to hold information which memory locations was added, edited or removed. All operations which may lead to change within some memory location has to store its identification in component called *change tracker*.

The change tracker is not meant to improve any inner operation of the snapshot or to reduce the complexity of the memory. Purpose of the tracker is to support memory algorithms which do not need to traverse the whole memory tree but only to check modified memory locations.

Laziness and change tracking are two major improvements of the new memory model implementation. Usage of these approaches dramatically reduced a time and a memory complexity of the memory model.

3.2.5.3 Parallel call stack

The original copy memory model identifies each level of a call stack by a call level of a snapshot. The disadvantage of this simple approach was found when the analysis started using a mechanism of *shared functions*.

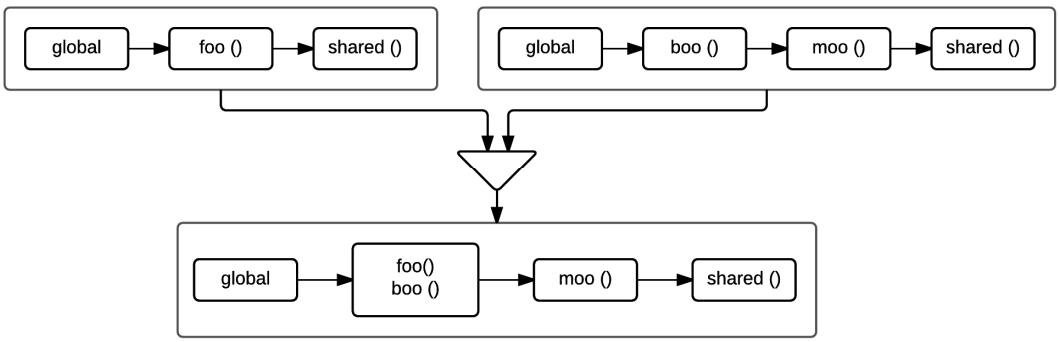


Image 3:14 Mixed local context in stack by call level

The shared functions have two basic usages – speed up the analysis of frequently used functions and allow the analysis to process recurrent functions. The full support of the shared functions is another feature of the new memory model which led to significant speed up of the analysis.

In the parallel stack each call level is identified by an ID of a program point graph of a called function. The advantage of this behavior is that the snapshot won't mix local contexts of distinct functions but only group contexts of the shared ones. The parallel call stack with the new merge algorithm improves precision of the analysis and brings faster convergence of the worklist algorithm.

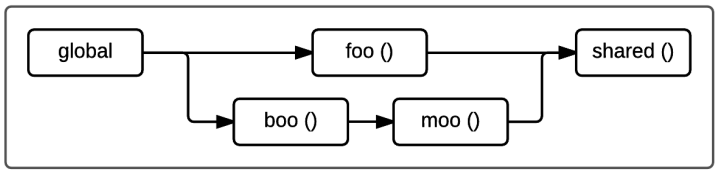


Image 3:15 Context separation in parallel call stack

The main advantage of the parallel call stack is that it simplifies the analysis of recurrent functions. Pure version of the incremental call stack caused that each call of recurrent function created new call level and gets analysis to an infinite loop. The original copy memory model tried to solve this problem – a call level of each snapshot could not be

changed. This solution was not ideal and a merge algorithm has a problem to determine which function contexts should be merged together. The parallel call stack is the ideal solution to prevent an infinite loop of recurrent functions. The merge algorithm may remain simple but be strong enough to separate complex function calls.

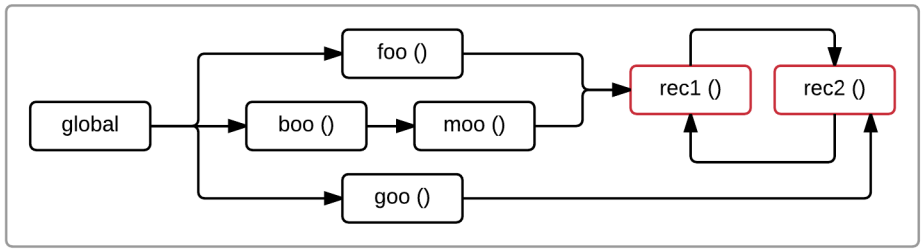


Image 3:16 The Parallel call stack – Indirect recursion starting from several functions

3.3 Memory model algorithms

The Weverca analyzer is not allowed to access an internal data of the memory model directly but use some operation within a public interface of the memory snapshot. An interface of the memory model contains set of available operations which can be used to provide the analysis of a statement in a processed program point. There are four distinct kinds of operations: read, update, flow and transactional.

Most operations are simple and can be processed directly by the interface of the snapshot. There are only few special algorithms to handle complex cases which may occur during processing of snapshot.

This chapter will describe main memory model algorithms – read, write, extend, merge and commit.

3.3.1 Memory path

A memory path describes how to traverse nodes of a memory tree by a BFS algorithm. Each memory path starts with a variable selection and continues by a sequence of indexes or fields. At the end of the traversing a memory path may be interpreted as a set of matching memory locations.

A memory path is very similar to a PHP access path described in the chapter 3.1 Memory location.

Definition of memory path

0. A memory path starts by a definition of a memory root and an identification of a source container within a memory stack
    - Variables *stackNumber.*
    - Controls *stackNumber.CTRL*
    - Temporary variables *stackNumber.TEMP*
    - Number of global level (zero) can be skipped
1. The rule 0 with a variable identification is a memory path
    - variables and controls are identified by *name identifier* *root $identifier*
    - Temporary variables are identified by ID *root $ID*
2. A memory path extended with field access is a valid memory path
    - Fields are identified by *name identifier* *memoryPath->identifier*

3. A memory path extended with an index access is a valid memory path
    - Indexes are identified by *name identifier* *memoryPath[identifier]*

A name identifier follows one of the three possible variants:

- Single name (direct name) *name*
- Multiple names (uncertain name) *{name1, name2}*
- Unknown (unknown name) *?*

A structure of a memory path can be illustrated on several examples:

- Global variable *$var*
- Local index *1.$arr[index]*
- Field at an uncertain index *$arr[{1,2,3}]->field*
- Unknown index of a local control *1.CTRL$control[?]*
- Temporary variable at the second level of a stack *2.TEMP$23*
- Unknown global variable *$?*

Final abstraction

Let P is a memory path. Then P consists of a stack number $P.stack$, length $N \in \mathbb{N} \cup \{0\}$ and a list of path segments P_0 to P_N.

- P_0 is a root segment with a *type of root container* and a *name identifier* (rule 1).
- P_1 to P_N represent index and field accesses with a *type of access* and a *name identifier*.

Interpreting memory path

When the analysis resolves a program statement which leads to a read or an update operation then a memory path is created. The memory model resolves each memory path as a description how to find a set of memory locations where to read from or write into.

A memory path is interpreted as a description for a BFS algorithm over the memory tree. The final product of the BFS algorithm is collections of *MUST* and *MAY* memory locations which fit the last segment of the memory path.

- A memory location in *MUST* collection is called *MUST memory location* – the path is called *direct;* locations have to exist in any possible flow.
- Location in *MAY* collection is *MAY memory location* – the path is uncertain, unknown or locations may be not be defined.

Note that each found location cannot be both *MUST* and *MAY*. When a new *MUST* location is created and a *MAY* already exists then the *MAY* location has to be deleted. Creation of a new *MAY* location is canceled when a collecting process already set the processed location as *MUST*.

An algorithm starts with resolving a stack level and processing a root segment of the memory path to resolves all matching *MUST* and *MAY* locations for the root segment. The main loop of the BFS algorithm iterates segments of the memory path and uses *MUST* and *MAY* locations from the previous iteration to construct new collections. After the last segment is processed then the algorithm prepares results and returns final collections of *MUST* and *MAY* locations.

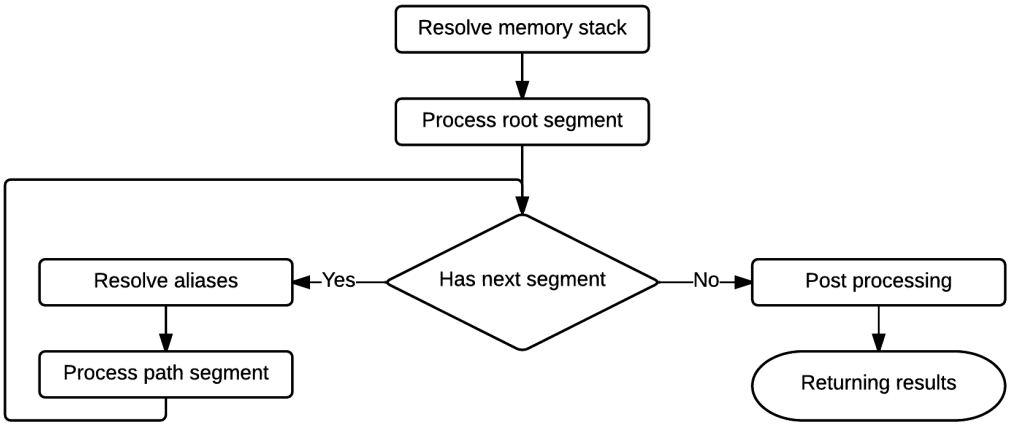


Image 3:17 The main loop of interpreting memory path by the BFS algorithm

The main part of processing each segment is to determine the type of access and to construct new collections of locations for the next iteration. The root segment is a special case because there are no input locations but only an identification of a variable container within the memory stack. The processing has to select a variable container and use it to resolve *MUST* and *MAY* memory locations matching the name identifier of the segment.

The processing of any other segment iterates previous locations and resolves next locations from the name identifier and array or object descriptors. An important part of the processing is possibility that some location may contain other value than an expected object or array. The presence of some values may be correct (index access over *string*, *any* or *undefined* values) or caused by an error in a PHP program. Processing has to create special memory locations for correct values or report the possible error to the user.

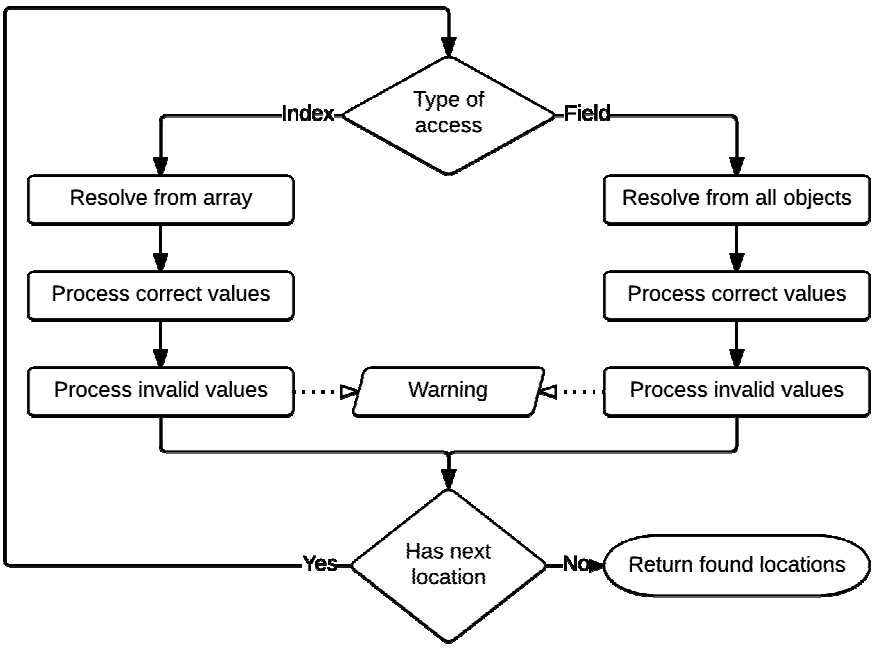


Image 3:18 The processing of an index or a field segment

Resolving memory locations

Locations are resolved from a name identifier and a selected variable container or a descriptor of an array or an object. The resolving process looks for any location which satisfies the name identifier of the current segment. The name resolving is driven by the type of the memory identifier: direct, uncertain or unknown.

The direct identifiers are resolved easily by looking for a memory location with the same name – the selected location may be inserted to *MUST* locations. The resolved location is considered as *MUST* only if the parent location is *MUST* and there is no uncertainty with the parent container:

- The parent container is one of variable containers
- The parent container is an array descriptor and the parent location has no other value
- The parent container is an object descriptor, the parent location contain only one object and has no other value

A location may be substituted with an unknown location if the current name is missing. In that case an undefined location always continues as a *MAY* location.

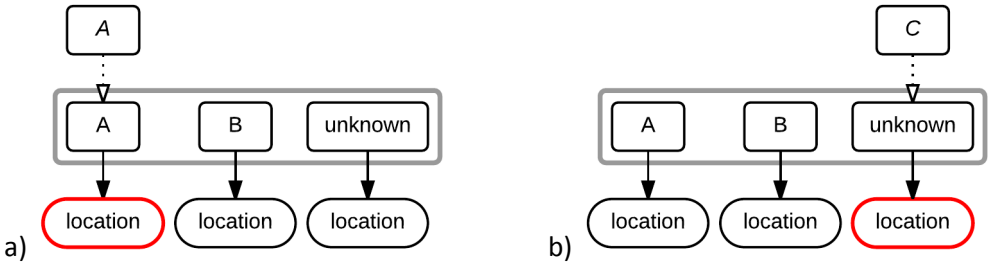


Image 3:19 Resolving locations for the direct path segment when location is a) defined and b) undefined

Resolving locations for an uncertain name is very similar to the previous case. The main difference is that there can be multiple names so the resolving process has to be applied to the all names. All matching locations are inserted into the *MAY* collection.

The last type of a memory identifier is an unknown identifier. This happens when the analysis is not able to compute dynamic value of an index or a variable name. In that case the memory model has to use all defined locations together with a special unknown location because any of them may occur the during program processing. All locations are inserted to the *MAY* collection.

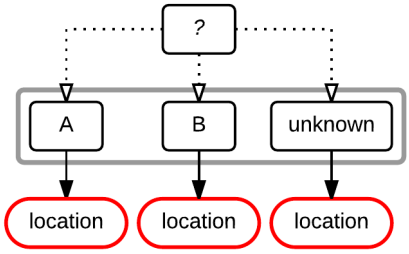


Image 3:20 Resolving locations for an unknown path

Note that a defined direct identifier is the only case when a resolved location may be considered as *MUST* but only if the parent location is *MUST* too. When a location once became *MAY* then all its ancestors has to be *MAY* too. A trivial consequence is that there cannot be any *MUST* location in a final output when a memory path contains any unknown or uncertain segment. Memory path on line 4 leads to *MAY* locations only:

```
1.  $directIndex = 1;
2.  $uncertainIndex = $condition ? 1 : 2;

3.  $arr[$directIndex][1];                  $arr[1][1]
4.  $arr[$uncertainIndex][1];               $arr[{1,2}][1]
```

Resolving alias references

Alias references are interpreted as edges between branches within a memory tree. Aliased locations have to be included by the BFS algorithm to ensure that a read or an update operation will process all relevant locations. Processing of aliases has to be written carefully especially for update operations.

The resolving process iterates all found locations and includes all *MUST* and *MAY* aliases to the *MUST* or the *MAY* collection. When a location is must then all MUST aliases are added into the collection of *MUST* locations and all *MAY* aliases into the collection of *MAY* locations. Otherwise when location is *MAY* then all aliases go to the *MAY* locations. Note that existence of a *MUST* alias is the only occasion when the collection of *MUST* locations contains more than one location.

The PHP has no limitation and it may happen that there can be an alias between any two memory locations. It may easily happen that there can be a cyclic reference on the processed memory path (caused by a cyclic alias or even an object reference). An existence of a cyclic reference means that any memory location can appear several times on the path. Algorithms have to be prepared even for that occasion.

3.3.2 Read

The analysis calls the read algorithm whenever a PHP code contains a statement which retrieves any data from a *PHP access path*.

The read algorithm is used when:

- Reading value from a variable
- Testing whether a variable is defined
- Iterating indexes of an array in foreach statement
- Resolving possible methods of an object for a method call

The read algorithm implements all read operations over the set of memory locations. The algorithm accepts a memory path and returns a set of requested values.

The read starts by collecting all memory locations where to get data from. The collecting process interprets given memory path by the BFS algorithm described in previous chapter. There is only one difference: the read operation does not need to distinguish between *MUST* and *MAY* locations.

The second part of the algorithm iterates found locations and creates an output set of requested data.

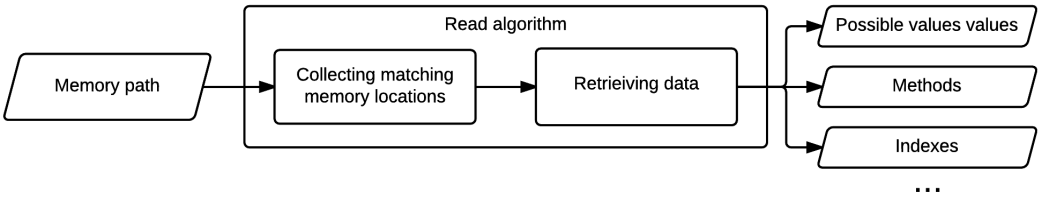


Image 3:21 Flow of The read algorithm

3.3.3 Assign

The analysis calls the assign algorithm when a PHP code contains a statement which updates value of some variable identified by a *PHP access path*.

The assign algorithm is used when:

- Assigning values
- Creating new alias
- Passing function parameters

The assign algorithm implements all update operations connected with a memory path. The simple assign algorithm accepts a memory path and a set of values. The result of the algorithm is modification of the snapshot – creating new locations, deleting overridden locations and storing new data.

At first the algorithm has to make copy of received data to prevent changes during the update of the snapshot. The next task is to find and prepare locations which will be updated and at the end finally perform an update of collected locations.

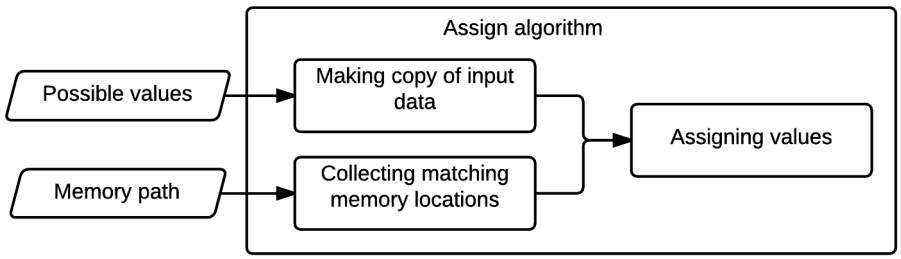


Image 3:22 Flow of the assign algorithm

There is no limitation of what can be assigned. The set of possible values is created by the analysis usually from an analyzed statement (result of read, code constant, new array or object) or as a result of an internal process of the analysis (assumption, control data …). The input may contain any number of primitive (number, string …), special (any, undefined …) or structural (objects and arrays identifiers) values.

Assigning primitive, special or even object values is very simple – just by inserting into the set of values of each target location. An array assign is special because of the copy semantic of arrays and limitation that each location can contain maximally one array. Whole branches of memory tree have to be copied and merged to the one when arrays are assigned.

There are some difficulties when the target location is placed within a sub tree of a copied array. In that case is very important to avoid changes of the source array when target location is updated. Otherwise there is a danger that the assign algorithm gives different result than the PHP assign operation. The assign may even get stacked in an infinite loop when trying to copy a grooving branch. The simplest solution to prevent a modification of the source array is to create a deep copy of the input data before any change is performed. This step may be skipped when there is no array to assign or the target location is not within the sub tree of the array.

```
1. $arr = array( … );
2. $arr[1] = $arr;
```

Collecting memory locations

The collecting process takes place right after the copy of the input data is performed. The collecting process translates the input memory path to the set of target *MUST* and *MAY* locations.

The main difference between the generic BFS algorithm and the write approach is when the algorithm finds a missing location, an array or an object. The assign algorithm has to be different because of dynamic character of the PHP:

- New variable, index or field is created on first assign.
- New implicit array or object is created when assigned to undefined location (location with undefined value)

The original BFS algorithm handles situation when the memory path contains a direct or an uncertain segment by continuing thru the unknown branch (se the Image 3:19). The write algorithm has the different approach. Image 3:23 illustrates situation when a memory location for the resolved name is missing. In that case a new location is created and the whole sub tree with all values is copied from an unknown to the new location. The newly created location is added to the *MUST* or *MAY* collection depending by the type of the segment and the parent location as usual.

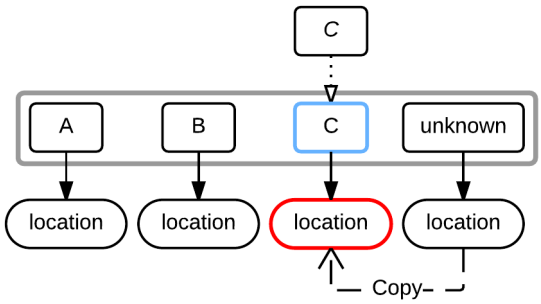


Image 3:23 Resolving a memory location when the location is undefined

The copying follows the character of an unknown location which is used any time when the analysis is not able to resolve the target name. In that case the memory model has to assume that the value may be inserted anywhere – even to a location which doesn't exist. Some PHP flows may lead to situation that there can be some value when the new location is created.

```
1. $arr = array( … );
2. $arr[$_POST[1]] = 1;                    $arr[?] = {undefined, 1}
3. $arr[$condition ? 1 : 2] = 2;           $arr[1] = {undefined, 1, 2}
```

Creating implicit arrays and objects is another difference between the original BFS algorithm and the assign approach. In the read algorithm missing array or object leads to propagation of an undefined value into the output. The assign algorithm has to be different because the PHP creates new array on the first assign into an index or new standard object in case of fields.

```
1. $arr[1][1][1] = 1;
2. $obj->fld = 2;
3. $arr[2]->fld->fld = 3;
```

The collecting process will create a new array when an index access is processed, the location contains no array and the location contains an undefined value. The collecting

process continues with processing the newly created array (typically by creating new index). The very same approach is applied to objects – new object has a generic PHP Object class.

Presence of an undefined value is very important. A location may contain primitive value and in that case an analysis warning is emitted and the location remains unchanged. Even when the location contains an index-able value (string or any) then no array is created (any value for objects). An interesting case happens when the location contains either an undefined or a non index-able value. In that case a warning is emitted but a new array (object) is created too. Note that an undefined value is removed after creation of an implicit array or an object if the processed location is *MUST*.

```
1. if (?) { $arr = 1; }                    $arr = { 1, undefined }
2. $arr[1] = 2;                            $arr = { 1, array }
                                           $arr[1] = { undefined, 2 }
```

The difference between an implicit array and object is handling situation when location contains either an array (object) or an undefined value. Location can contain only one array so no array is created in this case. Undefined value is removed and processing continues with created array.

```
1. if (?) { $arr = array( … ); }           $arr = { array, undefined }
2. $arr[1] = 2;                            $arr = { array }
                                           $arr[1] = { 2 }
```

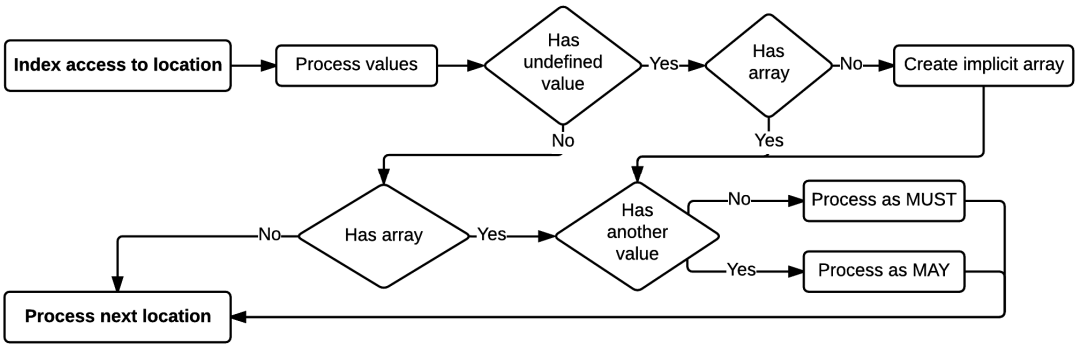


Image 3:24 Implicit array within an index access in the collecting process

There is no limitation how many objects may be stored within one location. A new object has to be created anytime when a PHP code access a field of an object which may be undefined. Note that this behavior may lead to the explosion of objects within the loop. In

that case an allocation site abstraction has to be use to prevent creating new object otherwise the analysis may stack in an infinite loop.

```
1.  if (?) { $obj = new Object(); }
2.  $obj->fld = 1;
```

```
$arr = { obj1, undefined }
$arr = { obj1, obj2 }
obj1-fld = { undefined, 2 }
obj2-fld = { undefined, 2 }
```

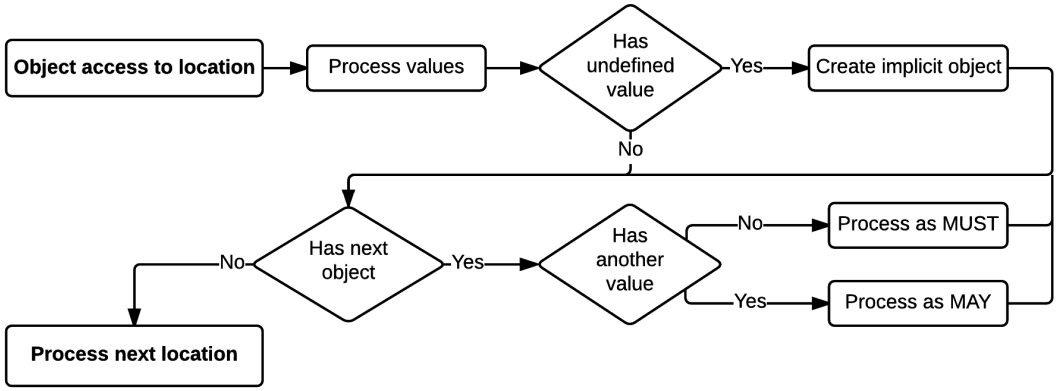


Image 3:25 An implicit object within a field access in the collecting process

Assign into collected locations

The collecting process ends when the last segment of the memory path has been processed. The assigning process will now write all given values into the collected locations. There are two distinct versions of the assigning which are determined by the type of a collected location – a *STRONG* assign for *MUST* locations and a *WEAK* assign for *MAY* locations.

The *STRONG* assign means that a content of the target location will be replaced with new values. The whole memory sub tree has to be deleted when the location contains an array. Primitive, special and object values are inserted into the target location. All arrays are merged to the new one which identifier is inserted to the set of possible values.

The *WEAK* update is different. The target memory location has to keep all of its values (even an undefined). Possible array have to be merged together with all sources and the target location will contain a product of merge and an undefined value has to be inserted to all cells of the new array.

Note that the merge process follows the same rules which are described in the chapter about the merge algorithm (see chapter 3.3.5 Merge). The only difference is that the merge

process has to group matching locations within different branches of the single memory tree.

Creating new aliases

The second variant of the assign algorithm is responsible for a creation of new alias. The analysis use this variant anytime when there is a new alias statement in an analyzed PHP source or when a function argument is passed by a reference.

The alias assign receives two memory paths – a source and a target. The first step is to run the collecting process for both memory paths. The next task is to assign all values from the source to the target locations (follows all rules described above). The final step is to create aliases between all combinations of source and target locations.

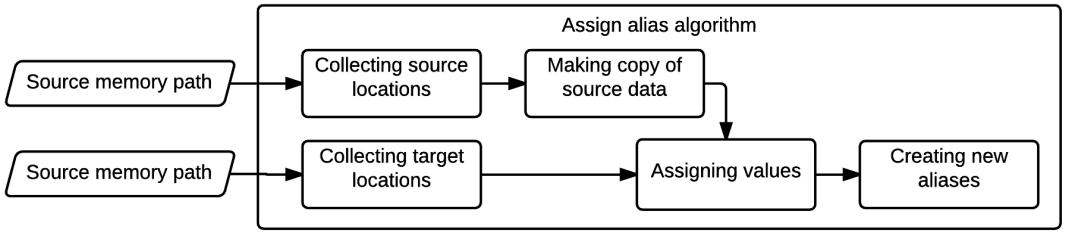


Image 3:26 Flow of the assign alias algorithm

When a target location is *MUST* then all aliases are removed and replaced by all sources. *MUST* source locations will become *MUST* aliases and *MAY* locations will be used as *MAY* aliases.

When a target is *MAY* then *MAY* aliases remains unchanged and all *MUST* aliases became *MAY*. All source locations will be used as *MAY* aliases.

Assign in second phase

The purpose of the second phase of the analysis is to compute special info flags for variables and values. The second phase uses data from the first phase (structural data and possible PHP values) and creates info values which are stored as possible values within the snapshot. Each algorithm must prevent any change within the structure and possible values from the previous phase.

The assign in the second phase is much simpler. There cannot be any object or any array value in a set of assigned or overridden values. Only info values are assigned so there is no

need for copying, merging or deleting branches of a memory tree. The main task is to collect target location but the final assign may remain very simple.

An approach of the original copy memory model was to use the same collecting process from the first phase. The only difference was that creation of new locations or implicit arrays and objects was deactivated. Note that the assign in second phase should never try to access a missing memory location. The reason is that each memory path in the second phase was already used in the first phase – all missing locations were created when the path was spotted at the first time.

The further development revealed that this approach was wrong. During the assign in first phase often happens that a new *MAY* location is created. In that case the collector uses values from an unknown location as initial values for the new one. In the second phase all locations have been already initialized. The old collecting process will never propagate all possible values from an unknown location.

```
1. $arr = array( … );
2. $arr[$_POST[1]] = $_POST[1];          $arr[?] = {undefined, any}
                                                 INFO {null, dirty}
3. $arr[$condition ? 1 : 2] = 2;         $arr[1] = {undefined, any, 2}
                                                 INFO {null}
```

The second phase of the analysis analyses the same statement on the same snapshot as the first phase. The same statement and the same values lead to the same memory path. The assign in the second phase do not need to run the collecting process but just use the data from the first phase.

The snapshot of the new memory model holds collecting data for each assigned memory path – a set of collected and newly created locations with sources. The collecting process in the second phase may be skipped and the assign process simply inserts values from all possible sources to all modified locations.

3.3.4 Extend

The Extend algorithm is used anytime when the analysis needs to copy all data from one the snapshot to another. The extend is usually first operation in a transaction before any other operation is processed. No update operation on extended snapshot can modify the previous one.

The extend algorithm is used:

- Within a single program point –from an input snapshot to an output snapshot
- Copying data into an input snapshot of a program point with a single parent

The extend algorithm accepts one source snapshot. The result of the algorithm is modification of the current snapshot which becomes a perfect copy of the given one – all data from the source snapshot are deeply copied into the target snapshot.

The extending is very common operation which is applied to nearly all snapshots and very often happens that it is the only modifying operation which is applied. The extend algorithm is the second most used memory algorithm (the first one is commit) so it should be very efficient and consume as low memory as possible.

Optimization of extend

Making a full deep copy of the source snapshot is not the best strategy. Even the original copy memory model tried to save resources by using immutable objects which might be copied by reference. But the main part of the snapshot (mainly inner associative containers) was still deeply copied on each extend. Even for statements which do not modify the snapshot.

Obvious improvement is to not copy anything when the snapshot won't be modified and copy just parts which will be affected by other operations. Sadly the snapshot has no information about future operations when extend is performed. It might have waited until the end of the transaction to apply all operations at once but the Weverca analyzer does not allow that – the result of an update has to be available immediately.

The new memory model solves all of these problems by adding laziness to the inner structure of the snapshot (as described in chapter 3.2.5.1 Data sharing). The extend algorithm just creates a new proxy object and copies a read-only reference to an inner data structure. The full copy is provided only if some update operation is applied. Even greater improvement may be achieved by composing an inner structure as a hierarchy of lazy objects. In that case each update operation will copy only affected parts.

Usage of laziness improved efficiency of the whole memory model. The extend is the quickest of all memory algorithms within the new memory model.

Call extend

The second variant of the extend algorithm is applied when the analysis is entering into a called function or a method. The call extend receives a snapshot, a *this* object and a program point graph of an entered function. The algorithm provides the normal extend and prepares a local context for the called function.

The copy memory model used a simple call stack – each call extend created a new call level with an incremental identification and set it as a local context.

At the beginning the new memory model was meant to reuse this approach and combine it with laziness. A creation of new empty local level is the only modification of snapshot. All other inner parts may be shared with the parent.

During the development was found that an incremental stack doesn't work well with shared functions. The original call stack was replaced by a parallel call stack which uses an ID of the called function PPG (see: 3.2.5.3 Parallel call stack). The extend algorithm of the new memory model tries to find and use an existing context in the current call stack or to create a new one with given ID.

Note that two distinct calls of the same function usually use two different PPGs. The same PPG is used only when the analysis resolves that the function is shared – the number of calls exceeds the defined limit.

3.3.5 Merge

The merge algorithm is used anytime when the analysis needs to combine data of several snapshots to a single one. The merge is always used (instead of the extend algorithm) to create an input snapshot of a program point with multiple parents. The merge is used:

- At the end of a conditional statement to combine data from all possible branches
- At beginning of the next cycle iteration to combine the beginning with the end
- In a function call to combine local levels of a shared function or results from uncertain calls (when the analysis computed more than one target)
- A jump statement – return, break, go to, exit, die, exceptions

The merge algorithm accepts a set of snapshots. A result of the algorithm is a modification of current snapshot to contain all possible structural data and values.

The merge algorithm iterates all locations in all source snapshots and identifies which source locations should be merged together. The process of identification is complicated because some memory location may exist in one snapshot but not in the other one. In that case the merge algorithm has to use data from some unknown location which may be propagated to the merged one.

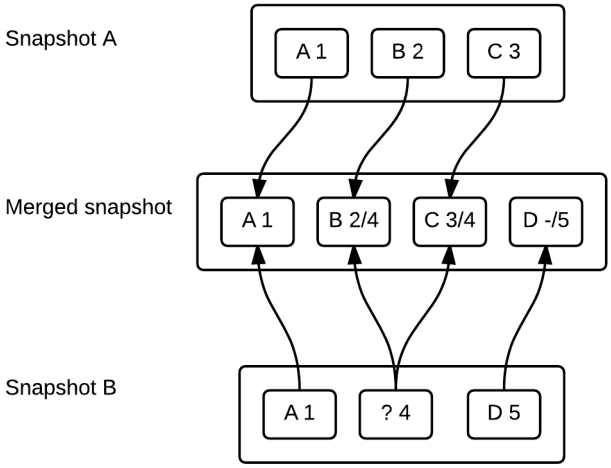


Image 3:27 The merge of two snapshots

The merge is very easy when a merged location is presented in all source snapshots. In that case the algorithm may use an identification of the memory location to get data directly from the source snapshots. The trickiest part of the merge is to find a suitable unknown location when the source location is missing. The main problem is that the source unknown location may be in a different branch of a memory tree – happens when the analysis is not able to count value of a higher index in a multi-dimensional array:

```
1. if ($_POST[1]) {
2.   $i = $_POST[2];                    $i = { any }
3.   $arr[$i][1][1] = 1;                $arr[?][1][1] = {undefined, 1}
4. }
5. else {
6.   $arr[1][1][1] = 2;                 $arr[1][1][1] = {2}
7. }
8. $v = $arr[1][1][1];                  $arr[1][1][1] = {undefined, 1, 2}
```

The problem of looking for locations is very similar to a collecting process of the read algorithm. The collecting process interprets given memory path by traversing a memory

tree from a variable root. The traversing process selects the memory location by an index if exists or an unknown location otherwise.

The merge can be imagined as a sequence of reads from all source snapshots and writes to the target. In that case the merge process starts by creating memory paths to all locations presented in all source snapshots. The next step is to use all distinct memory paths to collect source locations, gather all possible data and write them to the target snapshot. The collecting process ensures that the merge looks for a missing location in the nearest unknown branch if necessary.

The memory model does not use a described read/write approach. It is a great example to understand process of the merge but it would be very ineffective to use it in the real memory model. The copy memory model uses a BFS algorithm to traverse memory trees of all source snapshots simultaneously. The traversing starts in all variables and all objects and continues thru array branches only. A branch of each missing location (variable, field, and index) is substituted by an unknown branch. Note that each unknown branch is visited at least once – an unknown location itself has to be merged the same way as any other location.

The merge process starts with processing stack contexts with variables and object instances. The next phase traverses the memory tree to merge all locations and arrays. At the end the merge process will merge declarations of functions and classes.

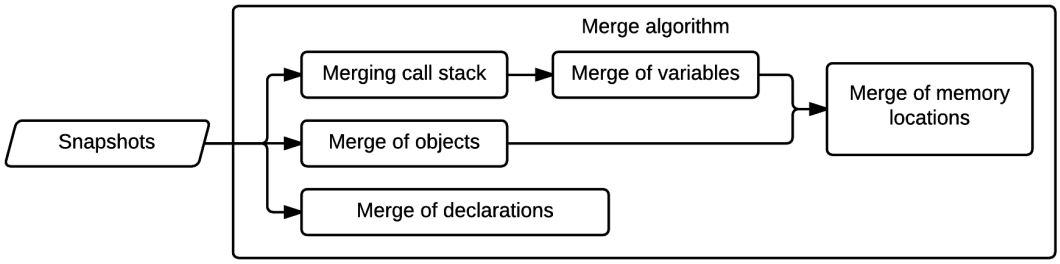


Image 3:28 Flow of the merge algorithm

Merge of call stack contexts and variables

The traversing process is driven by a queue of merge operations. Each operation contains an identification of a target location and a set of source locations for each snapshot (or information that some source location is missing – *WEAK* operation).

The merge begins with merging the memory stack and variables. The algorithm collects all possible identifications of stack levels. Then it selects the same levels from all source snapshots and creates new levels in the target snapshot.

The next task is to merge the same variable containers into the target. The algorithm creates a new empty container to collect all defined locations with distinct variable names (name identifiers). The next task is to create and enqueues a merge operation to schedule the merge of each target location (only the enclosing containers were merged so far). This is the place where the algorithm needs to determine whether to use an unknown index if a variable is missing in some snapshot.

The operation queue will contain one operation for all distinct variables in all variable containers. If the whole stack level is missing in some source snapshot then all merge operations from this stack level will be *WEAK*. Note that similar approach will be used even for merging object fields or array indexes.

Merge of object instances

The next task is to merge object instances on the heap. The reference semantics of PHP objects means that each object instance can be stored in several memory locations and each location can contain several objects. The memory model implements this behavior by auto-generated IDs which are used as references to object instances from memory locations. The semantic of the generated ID means that the merge can identify the same instance of an object in all merged snapshots. Note that this is not happening in case of arrys.

The merge process can use ID values to identify the same object instances across all source snapshots. The algorithm iterates all distinct ID values and merges matching object instances to the target snapshot. Fields of the same objects has to be processed the same way as variable containers in stack – collect possible fields by a field name and create merge operations (a missing object instance in some source snapshot causes that all object operations will be *WEAK*).

Note that objects instances are the same only if they come from the same *new* statement (or implicit creation). Two objects cannot be merged together when they were created in distinct branches of a code (even distinct iterations of cycle).

```
1. if (?)  $obj = new A();                    $obj = { obj1 }
2. else    $obj = new B();                    $obj = { obj2 }
3. $v = $obj;                                 $obj = { obj1, obj2 }
```

The target snapshot has no reference to any object instance so far. The references to the objects will be added during processing the merge operations in the next phase.

Traversing memory trees

The target snapshot now contains all possible stack contexts with variables and object instances with fields. No location has been merged but the queue of the merge operations is full of variable and field locations – the first level of memory locations within a memory tree.

The merge is now ready to do all merge operations and continue traversing the array branches. An iteration of the BFS algorithm provides a single merge operation and creates several more if any source location contains an array. The traversing ends when the queue is empty. The target snapshot will contain the full memory tree containing all possible locations and its values.

Any merge operation consists of three parts – merge of values, arrays and aliases.

The algorithm will take all possible values from each source location of a processed operation. All distinct primitive, special and object values can be stored into possible values of the target location. Each array value has to be removed and used in the next step. If the operation is *WEAK* then an extra undefined value has to be added into possible values of the target – the memory location can be undefined in some code flows.

The algorithm now holds the array values which were filtered out in the previous step. There can be only one array in any location – all possible arrays have to be merged to the one. The merge algorithm will use the same approach as before (variables and objects) to create a new array with all distinct indexes. If the current operation is *WEAK* or some array is missing then every created operation will be *WEAK* too.

The target memory location has to have all alias references which are present in all source locations. There can be either *MUST* or *MAY* aliases in source locations. Any *MAY* alias can be simply added to the target. A *MUST* alias can be created only if all source locations contain the same *MUST* alias reference and when the operation is *STRONG.* If the operation is *WEAK* or there is some location with missing *MUST* alias then the alias has to be used as

MAY. This approach will ensure that the alias remains *MUST* only if it has to be created in all possible flows of a PHP program.

Functions and classes

Functions and classes in the PHP can be declared conditionally. The snapshot has to contain definitions of all possible declarations. The same function or class can have multiple declarations.

```
1. if (?)  class A { };
2. else    class A { };
3. $obj = new A();                                  $obj = { obj1, obj2 }
```

The merge operation is quite easy because the PHP supports only a flat hierarchy of declarations. The algorithm has to collect all declarations from sources and stores them to the target. The target snapshot will contain two associative containers to map possible functions and classes by their name identifiers.

Call merge

The call merge is a special variant of the merge which is applied at the end of a function call – a complementary operation to the call extend. The call merge has to remove a local context of the finished function and renew an old local context. The call merge can receive one (more often) or multiple input snapshots.

Multiple snapshots mean that several distinct functions (or object methods) might be called in the single call statement. In that case the call merge has to combine results of all possible functions to the single snapshot which can be used as result of the function. There is nothing to merge when input snapshot is only one but an operation still needs to remove the old call level.

The original call merge (without shared functions) is quite simple operation which doesn't differ from the normal merge. The only difference is that algorithm will skip the local stack context. The same approach can be used even for a single snapshot – the merge will copy all content to ensure the snapshot separation.

The support of the shared functions brings more complexity to the call merge operation. This topic is described within this chapter in a section about the shared functions.

Change tracker and laziness

The original copy memory model always merged the whole memory model all the time. The whole memory tree was traversed and the merge operation was performed to all memory locations. The merge operation is performed even on locations which weren't modified in any conditional branch. Even a small *if* statement in the middle of a PHP code had caused that memory model had merged the whole unchanged state of the program.

```
1. // here is very long PHP code
2. if ($str == null) $str = "";
```

Unnecessary merges slows down the analysis and wastes a lot of memory resources – the mere operations modifies internal data which cannot be shared using laziness. Both problems could be solved if the merge would be able to identify changed locations since the beginning of a merged condition. The merge operation could be performed only on changes and the other parts could be shared.

The new memory model uses a change tracker to determine which locations were changed within the single snapshot. Another feature of the change tracker is that it has reference to the previous version of the snapshot. Note that there is no need to have a tracker instance if the snapshot was not changed – the next tracker will contain a back reference to the nearest modified snapshot.

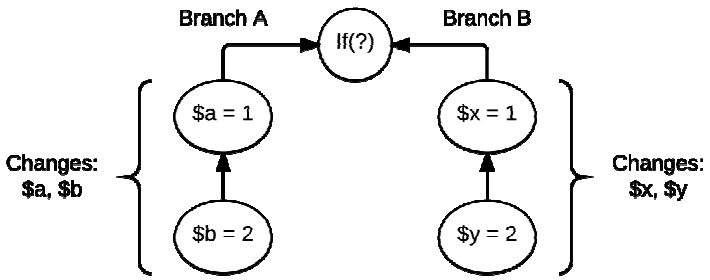


Image 3:29 Back references of trackers in conditional branches

The merge can use change trackers of source snapshots and look for the closest common ancestor using back references. This will lead the merge algorithm to find memory state right before all conditional branches. The algorithm may collect all changed collections between an end of a conditional branch and a located common ancestor. All changes together give information about possible differences between the beginning and the end of a conditional statement – other locations remains the same in all branches. The merge

algorithm is able to lazy copy the whole state of the common ancestor and to merge changes from all conditional branches.

The modified version of the merge algorithm starts with looking for a common ancestor and collecting of changed locations. Then it calls the extend algorithm to copy the common ancestor to the target snapshot – lazily copies all content. The merge process continues traversing the memory tree as before but only for modified locations. The algorithm starts with stack levels and objects which contains modified locations. All other stack levels, objects, arrays or locations are skipped and shared with the common ancestor.

Extending the common ancestor brings only one complication. There may be memory locations which were deleted in all conditional branches. All references and content of the missing locations must be deleted from the target snapshot – variables, indexes, arrays, aliases and values. The traversing process of the new merge algorithm needs to continue in all missing locations.

At the end the merge will set up the change tracker of the target snapshot. The back reference will be set to the common ancestor and the tracker will contain all collected modifications. The target snapshot will become a direct descendant of the common ancestor. The main effect is that any enclosing condition won't need to collect all changes from already processed branches but directly from product of their merge.

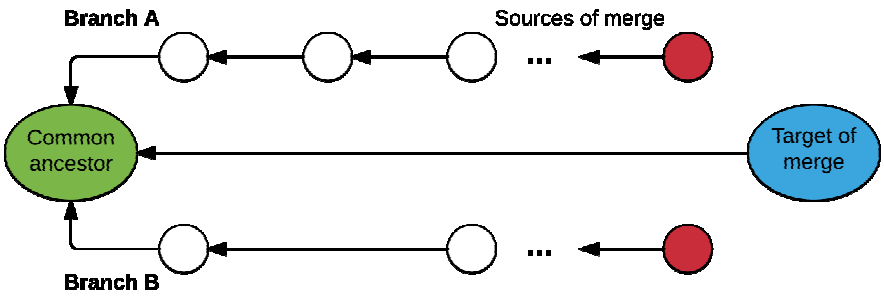


Image 3:30 Back references of memory trackers at the end of merge process

Shared functions

The mechanism of the shared functions is used as an optimization of a static analysis. When some function is visited very often then the analysis is able to group possible function parameters of distinct calls and analyze them together. This approach reduces a precision of the analysis because of propagation of results between function calls. However exact

values are less important than a behavior of the whole function. There is high probability that the function will behave very similar for distinct calls.

At the beginning the Weverca analyzer provides a new analysis for each function call. The shared functions are used when the number of calls of some function exceeds the specified limit. In that case the Weverca will collect all input snapshots, merges them together, provides the analysis of the function over merged snapshots and uses result of function in all call points. Any next function call will merge all snapshots again and check whether the product of the merge differs. If the merged snapshot is the same then the Weverca will use the last output or provides new the analysis if there is any difference. There is high probability that the merge products stop differ in a few calls (because of a widening and a simplification).

From the memory model point of view is important to provide correct merge operation at the beginning and the end of the shared function. The memory model receives set of snapshots at the beginning of the shared function. Each snapshot is result of some *call extend* operation – already contains a call stack with the current local level and values of all function parameters. The main problem for the merge algorithm is to group proper stack levels – especially local.

At the end all call points will receive the same final snapshot. Each call point has to run its call merge operation to remove all extra stack contexts and to merge data from other possible calls.

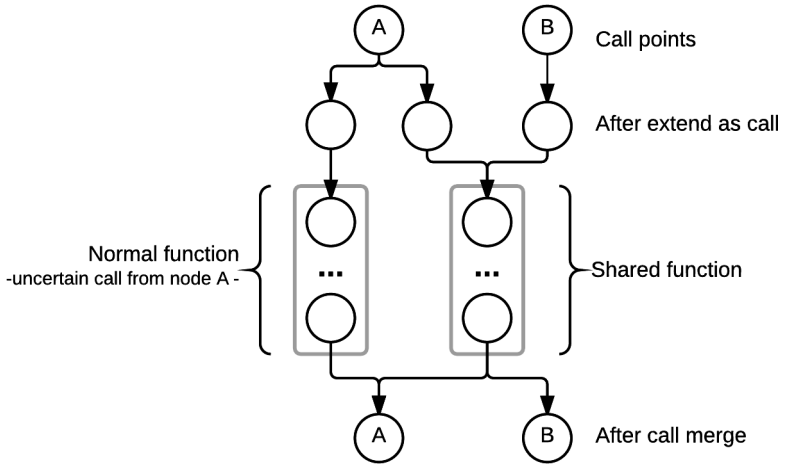


Image 3:31 The sequence of program points in the shared functions

The original copy memory model implemented the shared functions with the incremental call stack. The merge at the beginning of the function selected the highest call level – all local stack contexts were merged to the new local level with the highest number. All other stack contexts were merged by their call level. The call merge at the end of the shared function trimmed all extra stack contexts and merged all data to the final snapshot.

Change tracker in shared functions

An approach of the original copy memory model has two major issues – mixing stack contexts of distinct functions and merging unused parts of a snapshot with shared values back to a call point. Problem of merging function contexts is easily solved by the parallel call stack (explained in chapter 3.2.5.3 Parallel call stack).

The issue of merging unused data follows basic character of the shared functions. A context of a shared function is often the product of the merge of very different states of a PHP program. The analysis needs to merge everything to be able to read all possibilities from all call contexts at the beginning of the shared function. The problem is that merging leads to propagation of all values between all call points at the return from the function. The snapshot at the call point will contain data from remote snapshots which reduces the precision of the analysis or brings false positives.

The parallel call stack solves this issue for the data in the function contexts. The separation of stack levels prevents mixing context of distinct functions however the mechanism of the merge still causes that an undefined value will be added to each location and propagated back to the call points. The call stack is also not able to prevent merging of global contexts which are present in all snapshots.

The final solution is to skip all unmodified data in the call merge. Call merge at the end of the shared function has to use only those locations which were changed during the analysis of the function. The new memory model is able to use the sequence of change trackers to collect all changed data in the single function call.

The merge operation in the new memory model uses the change tracker to find the nearest common ancestor and to collect all changes within the memory model. The merge at the beginning of the shared function uses the same approach. The merge operation finds the common ancestor of all call points and collets all changes between them.

The call merge will receive snapshots from all possible function calls. The common ancestor is always the snapshot of the call point when there is no shared function. When there is

even one shared function then the common ancestor can be very distant – at least the same ancestor as at the beginning of the shared function. The collecting changes would have propagated lots of unused values to the call point.

The final approach of the call merge has to be different. The ancestor of the merged snapshot is always snapshot of the call point and not the common ancestor of the sources. The operation still looks for changes but looking has to be stopped before the beginning of each possible function is reached. If looking for data stops between merged snapshots at the beginning of the shared function then only the modified locations will be collected. Unchanged data won't be propagated but used directly from the parent call point.

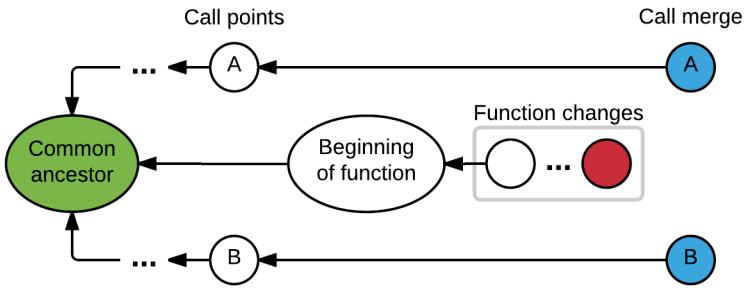


Image 3:32 Back references of memory trackers at the end of the shared function

3.3.6 Commit

The analysis calls the commit algorithm at the end of each transaction. The commit takes no input parameters and returns a Boolean value to determine whether the snapshot was changed during the transaction or not. The analysis uses this information to determine whether the snapshot was changed due to previous iteration of the worklist algorithm. The commit algorithm also performs all necessary operations connected with closing of the transaction.

The commit is very important for the worklist algorithm which uses it to determine whether a fix point was reached. Any error of the change detection may confuse the analysis to stack in an infinite loop or to skip some possible states of an analyzed program. Each snapshot has to keep state from the beginning of the transaction and compare it with state at the end (when the commit is performed).

The original copy memory model simply iterated all possible locations of the current and the previous state. The Commit algorithm applied a value simplification (removing duplicate values) and determined whether there was any change within the transaction. The problem

is that the commit is the most used algorithm in the memory model. The algorithm is called at the end of each transaction – even on transactions without any update operation. Looking for changes in unchanged snapshots is one of biggest time wasting issue of the original memory model.

The commit doesn't need to be performed when there was no operation within the transaction because this information can be obtained from the previous nearest modified snapshot in program flow. When there is any update operation then the full commit has to be performed to determine whether the update changed the state of the snapshot. An information about change can be stored within the snapshot internals and distributed during the extend algorithm. A snapshot without any update can be modified due to the previous iteration only if some update operation in the parent snapshot changed its state. When no update was performed the commit won't check the snapshot but will return stored value immediately.

Another improvement is not to use the commit after the first transaction of the snapshot but simply return information that the snapshot was changed. Only issue is that the original commit was meant to simplify a set of possible values of modified locations. The simplification removes repeated or mandatory values to keep memory model as small as possible. The simplification is important but there is no special reason that it has to be part of the commit algorithm. The new memory model applies simplification on each modification of possible values which keeps memory model small and allows skipping the commit algorithm on the first run.

Finally the commit algorithm doesn't need to check the whole snapshot whether there is any change but to use a change tracker to test only modified locations. Commit is also able to remove unchanged locations from the tracker to reduce complexity of the future merge operation.

All three improvements together caused that commit is now a very efficient algorithm in the new memory model.

4 New memory model

The purpose of the new memory model is to solve all main issues of the original copy memory model. The new memory model has to follow the same main principles as the original one. The main functional difference is that the new memory model has to be able to change of all internal data storages and memory algorithms. The modularity brings possibility of comparing several techniques of optimization to clarify which methods bring the best improvements.

The name of new memory model is *modular copy memory model*.

Following chapters will describe a final implementation of the modular copy memory model. Note that only the main principles and classes will be described. Please visit documentation or a source code to get more familiar with implementation details.

4.1 Modular implementation

The new modular memory model was built on the basics of the original copy memory model from first release of the Weverca analyzer. The first version of the modular memory model tried to re-use as much original code as possible but refactor a package structure and divide data and algorithms. A further development led to a creation several variants of data storages and memory algorithms.

A process of implementation of the modular memory model was split to several stages. The first stage was to identify logic parts of the memory model and organize old classes into the new package structure.

The second stage came with modularity and definition of interfaces of the memory storage and the algorithms. The memory snapshot class stopped providing the memory algorithms directly but started using the modularity and the interfaces. The memory model wasn't working after this stage but was prepared for adding new variants of the memory storage and the algorithms.

The third stage was to refactor an old implementation to use new interfaces. All algorithms had to be removed from the snapshot and memory storage and encapsulated to a single entry point. The algorithms had to be modified to not to access memory storage directly

but to use the new interface. The memory model provided the same functionality as the original implementation at the end of this stage.

The final fourth stage was responsible for creation of optimized variants of the memory storage and the algorithms. A lazy storage, a change tracker and optimized algorithms were created during this stage.

4.1.1 Structure of modular copy memory model

The main C# solution of the Weverca analyzer currently contains three distinct memory models: *virtual reference memory model*, *copy memory model* and *modular copy memory model*. The source code of all three memory models can be found in the C# project *Weverca.MemoryModels*.

A user of the analyzer framework has to provide a factory which creates an empty snapshot of the selected memory model. An abstract class *MemoryModels* contains prepared factories for all three types.

Whole code of the new memory model is located in the folder *ModularCopyMemoryModel* with a following namespace hierarchy:

Modular Copy Memory Model

Weverca.MemoryModels.ModularCopyMemoryModel

It is a root namespace for all parts of the new memory model. The root namespace contains the main memory model class *Snapshot* and a list of implementations variants. Each variant contains a set of factories to create a new snapshot instance and all modules of the memory model.

Snapshot Entries

Weverca.MemoryModels.ModularCopyMemoryModel.SnapshotEntries

This namespace contains an implementation of the Weverca interface for memory paths. Snapshot entry classes are used to build a path and to provide a read or an update operation over the snapshot instance.

This is the place where the read and the assign algorithms are invoked.

Memory

Weverca.MemoryModels.ModularCopyMemoryModel.Memory

The memory namespace contains all common classes used to represent a memory state. The namespace contains classes which identify memory locations or defines memory paths. There are also containers to store data of algorithms within the snapshot to speed up run of the assign and the merge algorithms.

Interfaces

Weverca.MemoryModels.ModularCopyMemoryModel.Interfaces

This namespace contains the definition of interface for the memory storage and the algorithms. The namespace consists of sub namespaces *Algorithm*, *Data*, *Structure* and *Common* (interfaces for the memory algorithms and the memory storage split to a structure part and a data part). The *Structure* namespace contains interface of whole structural storage and its inner objects and containers. The *Common* namespace contains shared interfaces for internals of the data and the structure containers.

The namespace contains a *ModularMemoryModelFactories* with a *MemoryModelFactory* instance and other factories used by the modular memory model.

Implementation

Weverca.MemoryModels.ModularCopyMemoryModel.Implementation

The implementation namespace contains implementations of interfaces defined in the previous package. Contains sub namespaces *Algorithm*, *Data*, *Structure* and *Common*. The *Algorithm*, *Data* and *Structure* namespaces are split to more sub namespaces by a variant of an implementation. The namespace *Common* contains classes which are used in several distinct implementations.

Logging

Weverca.MemoryModels.ModularCopyMemoryModel.Logging

This namespace contains all classes responsible for a logging, a benchmarking or a generating the visual representation of the memory model.

Utils

Weverca.MemoryModels.ModularCopyMemoryModel.Utils

The utils namespace contains utility classes with common operations. The utility classes are used across the whole memory model.

4.1.2 Basic interface

Snapshot class

A class *Snapshot* is the main class of the modular copy memory model. It is derived from an abstract class *SnapshotBase*. The snapshot class has no public constructor – instances have to be always created using some instance of the *ModularMemoryModelFactory*.

Each instance represents the whole memory state at some point of a PHP program. The class contains high level description of the memory state, references to a structural and a value storages and entry points of all memory operations. The Weverca analyzer creates a new instance with the factory object and calls the public interface of the *SnapshotBase* to invoke a memory operation.

The original copy memory model contains the similar *Snapshot* class. Difference between the original and the new *Snapshot* is that most functionality was moved from the main class to the memory algorithms.

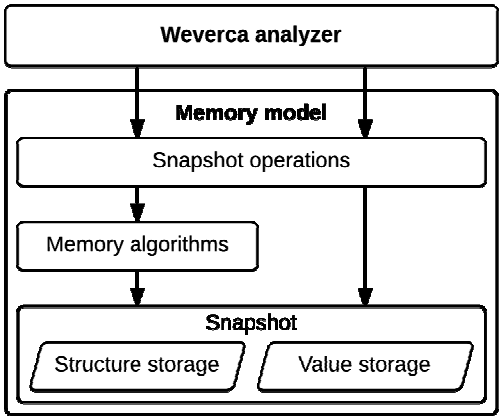


Image 4:1 A layer architecture of the snapshot object

Memory Index

A *MemoryIndex* is an immutable class which contains information to identify some memory location within the snapshot. Each instance has to be able to be compared with another one to determine whether they point to the same location. The most common usage is to be an index within some associative container to identify data related to a memory location.

The concept of memory indexes came from the original memory model. The memory index solves the problem to determine that two memory locations are the same even if they were created in distinct branches. The snapshot cannot generate a unique identifier when the memory location is spotted at the first time because two snapshots in different branches have no connection with each other. The memory model has to use the full memory path of the memory location to be able to clearly define each memory location. Note that a memory path starts by some variable or field of an object and follows with sequence of index accesses.

Each instance of the *MemoryIndex* contains the whole memory path from the root of a memory tree to an associated memory location. If two indexes contain the same memory path then they are the same. A proper implementation of methods *Equals* and *GetHashCode* allows using indexes as keys in standard associative containers. The implementation of the memory index overrides both methods to compare memory paths of two instances or to generate hash code from the memory path.

Snapshot entry

A snapshot entry is a special object created by the snapshot when the analysis needs to operate with a memory path. The analysis is able to use the public interface of the Snapshot to create a snapshot entry which represents some root variable. An interface of the snapshot entry allows to continue the access path by an index or a field access or to provide some memory operation on the created path.

The modular memory model contains two types of snapshot entries. A basic implementation is in a class *SnapshotEntry*. Instances of the *SnapshotEntry* always contain some memory path which starts in some variable. A class *SnapshotDataEntry* provides special behavior when the analysis needs to create a memory path to given *MemoryEntry*. This memory entry can contain values from several locations. The memory model needs to merge all possible arrays to the special temporary location and use it as a root of a new

access path. The *SnapshotDataEntry* provides the same operations as the normal SnapshotEntry.

The memory path is represented as instance of a class *MemoryPath*. This class follows the same pattern as the *MemoryIndex*. The difference is that the *MemoryPath* can contain a field access and each segment of the path can have several name identifiers as was specified in definition of a memory path in the third chapter (3.3.1 Memory path).

Modular behavior

The one of the main requirements to the new memory is ability to switch between implementations of memory storages and algorithms. The modular architecture simplifies a future development of the memory model – future developers are able to change an algorithm or an inner container without affecting the other parts. It is also extremely useful for a benchmarking and a comparing several variants of implementation. The final release of the Weverca analyzer will use the best combination of the memory storage and the algorithms but for a purpose of this text is necessary to measure and compare all main variants.

The modularity of the new memory model is achieved by factory objects which create new instances of all main parts. A new factory instance for each module has to be passed to the constructor of the *ModularMemoryModelFactories* object. An instance of factories is distributed across whole memory model by *create* method in factory objects. The *ModularMemoryModelFactories* object contains:

MemoryModelSnapshotFactory is passed to the Weverca analyzer to create a new instance of the *Snapshot* class. An instance of this factory is created in constructor of the *ModularMemoryModelFactories* and cannot be changed. Enclosing factories object will be passed to the constructor of the snapshot which starts to use it to invoke any factory or an algorithm.

SnapshotStructureFactory and ***SnapshotDataFactory*** create new instances of memory structural and value storages. The merge and the extend algorithm should call these factories to create new instances of memory storages and pass them to snapshot.

StructuralContainersFactories contains a set of factories to create internal parts of the memory storages which belongs to the proper variant. The algorithms and the storages should use these factories to create new instances instead of creating instances directly using the *new* operator.

MemoryAlgorithms and ***InfoAlgorithms*** are not factories but contain instances of algorithms for proper phase of analysis. Each algorithm has to be stateless so the Snapshot doesn't need to create new instances but use the same again. The memory model will decide whether to use memory or info algorithms by the current phase of the analysis.

Benchmark and ***Logger*** are not factories but singleton instances. These instances are used by the Snapshot to perform logging and measuring of the algorithms and the operations. The default instances are empty and provide no functionality but might be replaced by a custom implementation when needed.

There are several instances of the *ModularMemoryModelFactories* which contains a set of factories and the singletons which belongs to the same variant. The prepared factories can be found in a *ModularMemoryModelVariants* class which can be used as example how to create a new variant. A *DefaultVariant* property is used to set a variant which will be used as the Modular memory model within for the Weverca analyzer. The other variants have to be manually selected by passing an instance of the *MemoryModelSnapshotFactory* to the constructor of the analyzer.

4.1.3 Memory storage

The memory storage is a part of a snapshot instance which is responsible to store all structural data and values as was described in the third chapter. It consists of structure and value storages which are part of each snapshot instance. For more information please visit the chapters 3.2.2 Structural data and 3.2.3 Value storage.

Structural storage

An interface of the structural storage for the memory model is defined within the namespace *Interfaces.Structure*. The namespace contains several interfaces which define all basic elements of the memory tree as described in the third chapter. There are interfaces for the whole structural storage itself, stack contexts, object and array descriptors or internal data container for storing values describing a memory location. An implementation can use its own approach but has to provide the requested interface to be able to work with the rest of the memory model.

The whole structure is encapsulated to a single object which implements interfaces *IReadOnlySnapshotStructure* and *IWriteableSnapshotStructure*. The interfaces defines read and write methods to access a stack levels with variables, created arrays, created objects,

definitions of memory locations and declared functions or classes. The other parts of the memory models use these methods to provide a requested operation on with the structural data.

Most of the instances of all memory tree objects are created in the snapshot class or in the one of the algorithms. The structure is responsible only of creating the stack levels, the index definitions and inner containers (associative maps and sets). It is highly recommended to use the *StructuralContainersFactories* from the memory model factories object to create the inner instances if necessary.

For example the list of array descriptors should be stored in an associative container. If a structure uses an appropriate factory then there is possibility that some variant may use more efficient implementation than the default one. Associative container has to implement *IReadonlyAssociativeContainer* and *IWriteableAssociativeContainer* defined in the namespace *Interfaces.Common*. The instances of the associative container are created using *IAssociativeContainerFactory*.

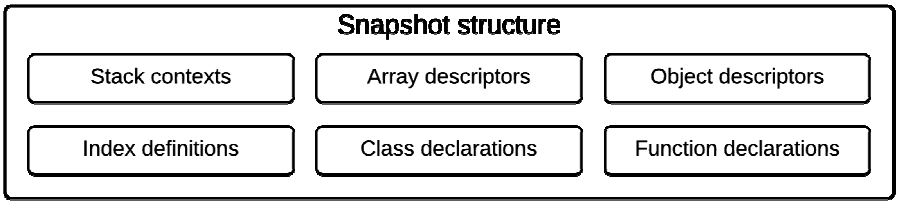


Image 4:2 The block diagram of the snapshot structure object

Stack context

A memory stack has to be implemented as an associative container which maps a numeric call level to an object with a stack context. Each stack context has to implement *IReadonlyStackContext* and *IWriteableStackContext*. These interfaces requires that the stack level has to provide an access to a collection of declared variables (variables, control variables and temporary variables) and a collection of arrays which belongs to selected stack level. Instances of the stack context class are always created within a snapshot structure object but it should use a factory instance of *IStackContextFactory* class.

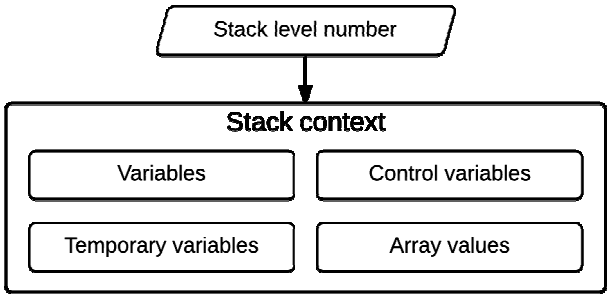


Image 4:3 A block diagram of the Stack context identified by a stack level number

A Collection of variables is an associative container which maps a string variable name to a memory index and holds a memory index of an unknown memory location. The variable container has to implement *IReadonlyIndexContainer* and *IWriteableIndexContainer*.

The memory stack with the variable containers is used to retrieve information about used call levels and declared variables. There is no special factory to create an index container. An implementation of stack context is fully responsible to provide its own index container.

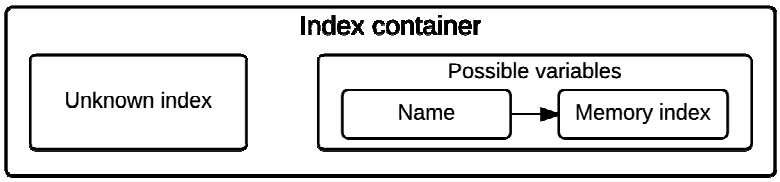


Image 4:4 A block diagram of index container with an associative array of variables

Descriptors

Arrays and object descriptors are stored in associative containers which maps an array or an object value to a description object with a list of possible indexes or fields. An array descriptor is a class which implements interfaces *IArrayDescriptor* and *IReadonlyIndexContainer.* Its purpose is to map index name names to memory indexes. An object descriptor follows the same pattern – interfaces *IObjectDescriptor* and *IReadonlyIndexContainer*.

Both descriptors are immutable classes which can be modified only using special builder instances. The builder instance has to implement *IArrayDescriptorBuilder* or *IObjectDescriptorBuilder* and *IWriteableIndexContainer*.

An instance of new descriptor is created in the snapshot or an algorithm using *IArrayDescriptorFactory* or *IObjectDescriptorFactory*. The instance of the snapshot structure doesn't create new ones but just holds the collections of descriptors

Index definition

Defined memory locations are stored in another associative container which maps memory indexes to index definitions – interface *IIndexDefinition* with builder *IIndexDefinitionBuilder*. An index definition contains all structural data related to a memory location – aliases (*IMemoryAlias*), possible object references (*IObjectValueContainer*) and a possible array value.

An empty index definition instance is created by the Snapshot Structure when index is inserted at the first time or in the merge algorithm. Both places should use *IMemoryAliasFactory*. Inner values can be left empty or created by *IMemoryAliasFactory* or *IObjectValueContainerFactory*.

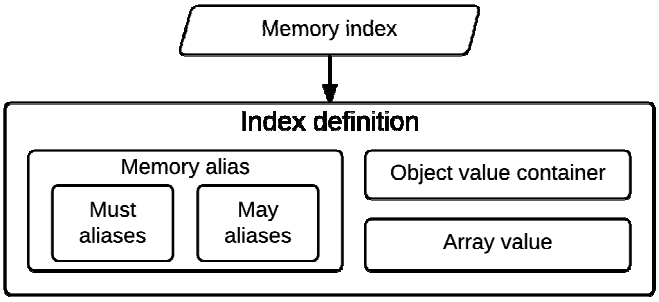


Image 4:5 A block diagram of an index definition object

Declarations

The last part of a structural object consists of two associative arrays with declarations of functions and classes. Both containers map a *QualifiedName* to a set of *FunctionValue* or *TypeValue* objects. The declaration container should use generic interfaces *IReadonlyDeclarationContainer* and *IWriteableDeclarationContainer*. Instances are created within the snapshot structure by *IDeclarationContainerFactory*.

Snapshot Data

The memory value storage is much simpler than the structure storage. All values are stored within a single object implementing *IReadOnlySnapshotData* and *IWriteableSnapshotData*. Both interfaces define behavior of an associative container which maps memory indexes to

instances of a class *MemoryEntry*. Memory entry is immutable class defined in the Weverca framework to hold all possible values for some memory location. The snapshot data should use an interface *IAssociativeContainerFactory* to create an inner associative container with values.

Proxy classes

The snapshot instance doesn't hold the structure and the data containers directly but uses a special proxy object implementing *ISnapshotStructureProxy* and *ISnapshotDataProxy*. These objects contain a reference to the single structural or the data container. The proxy object doesn't use any factory but creates new instances of the inner container directly calling its constructor. The instances of proxy objects are created by *ISnapshotStructureFactory* and *ISnapshotDataFactory.*

The snapshot instance contains one instance of *ISnapshotStructureProxy* and two instances of *ISnapshotDataProxy* – for memory and info phase of the analysis.

The snapshot structure is an entry point to the memory container. Any algorithm which wants to operate with the container has to declare whether it will modify the container or just read existing data. This behavior allows proxy to prevent modifications when the snapshot is not allowed to change (it is out of a transaction or in a wrong state). Any user of a proxy can access inner container using its properties *Readonly* or *Writeable*.

The proxy object is also great place for implementing a lazy behavior of the memory containers. The new copy of the proxy object won't create the copy of an inner container but shares a read-only instance. The structure is copied when some algorithm requests a writeable variant of the inner container.

4.1.4 Memory algorithms

The new memory model defines interfaces for the several algorithms. There are all main algorithms from the third chapter and two others providing tasks for the memory model. Each algorithm has to implement a proper interface and provide a factory class implementing *IAlgorithmFactory<T>* – *T* is the interface of the algorithm. The memory model creates a single instance of all algorithms in constructor of the *MemoryModelFactory*. Note that an algorithm has to be stateless – a single instance is called from all snapshots.

An algorithm implementation can be split to a memory and an info part. The snapshot will use a proper version of the algorithm by the current phase of the analysis. If some algorithm supports both phases then there will be two instances and its factory has to be used either for a memory and an info algorithms.

All algorithm interfaces are defined in the file *Algorithm.cs*.

Assign algorithm

The assign algorithm has to implement interface *IAssignAlgorithm*. The algorithm has to implement operations *Assign*, *AssignAlias* and *WriteWithoutCopy* which are called from an instance of a Snapshot entry.

The *Assign* and the *AssignAlias* follow an approach which was described in the third chapter (see chapter 3.3.3 Assign). The operation *WriteWithoutCopy* writes given memory entry directly to the target locations without checking constraints of the memory model. This operation can corrupt the state of the memory model if target location contains an array or an object. The analysis should use this operation only if the target location doesn't contain any structural value.

A memory version has to interpret a memory path and locate all target locations. Missing locations, objects and arrays have to be created. Given value has to be written to variable storage. An info version mustn't modify the structure but just write to existing locations in the info value storage. A missing location causes an exceptional state.

Read algorithm

The read algorithm has to implement *IReadAlgorithm*. The interface defines methods *Read* and *IsDefined* which accepts a snapshot and a memory path. The *Read* method returns a memory entry with all possible values from all locations of given path. Method mustn't change the snapshot but find all matching locations or add an undefined value when is missing. Version for both phases can be the same but an info phase has to return possible values from an info storage. The *IsDefined* returns a Boolean value whether all path locations are defined in the structure.

Other interface methods accept a snapshot and a memory entry from the *Read* algorithm to provide other read operations.

Commit algorithm

An interface *ICommitAlgorithm* defines methods *CommitAndSimplify* and *CommitAndWiden* for the commit algorithm. The commit algorithm is called directly from the snapshot instance when the analysis requests an end of the transaction. Both methods accept a committed snapshot and limit for the number of values in a memory entry to apply simplification.

The *CommitAndSimplify* has to locate all changes in the snapshot and run basic simplification which removes duplicated values from modified entries. The method returns a Boolean value indicating whether the snapshot has changed. The memory version looks for structure and value changes. The info version can process only info value storage.

The *CommitAndWiden* provides the same operation as the previous method but applies a widening to every modified location. The widening operation provides an advanced simplification which is able to group several simple values to the one aggregated value. The widening can group multiple numbers to single interval or replace all possible values with the single *AnyValue*.

Note that the memory model does not implement the widening and the simplification. These operations are provided by an instance of the *MemoryAssistantBase*. The analysis framework is responsible to pass the memory assistant to each snapshot instance.

Merge algorithm

The merge algorithm is defined in an interface *IMergeAlgorithm* which groups the extend and the merge memory algorithms. The algorithm is called directly from the snapshot when the analysis needs to perform some extend or merge operation.

The extend methods are *Extend* and *ExtendAsCall*. Both methods take a source and an extended snapshot. The extended snapshot will became a perfect copy of the source one. The *ExtendAsCall* takes two extra parameters with a function PPG and a possible *this* object. At the end of the *ExtendAsCall* the extended snapshot will contain a copy of the source and a prepared local level for a function call.

The *Merge* method is responsible to provide the basic merge of several snapshots to a given target snapshot. A memory version should provide full merge as described in the third chapter. An info version cannot change the structure but just merge data in info value storage. A missing memory location will cause an exceptional state.

The algorithm contains several advanced merge methods. The *MergeAtSubprogram* is called at the beginning of a function to merge several products of the *ExtendAsCallI* and prepare a context of a shared function. The *MergeWithCall* has to merge data from a function context back to a call point. The *MergeMemoryEntry* is responsible for merging data from given memory entry into the new temporary location. Last method is called from an instance of the *SnapshotDataEntry*.

Memory algorithm

The memory algorithm has to implement an interface *IMemoryAlgorithm* which contains several methods to modify internal parts of the memory storage. The memory algorithm is called from the snapshot or another algorithm to provide a common operation over the snapshot.

The *CreateMemoryEntry* is used to create new memory entry with given set of values. The method is able to provide a basic simplification (i.e. remove repeating or duplicated values). Other algorithms should use this operation instead of creating a new memory entry. The *DestroyMemory* drops whole branch of a memory tree which starts at given location. The *CopyMemory* copies whole branch from given location to the new one.

Print algorithm

The algorithm is defined in *IPrintAlgorithm* interface. Print algorithm has the only method *SnapshotToString* which is responsible to build a text representation of given snapshot. The algorithm is called from the snapshot to write a whole memory state as an output of the analysis or to a log file.

4.2 Variants of implementation

The previous chapter described the internal parts of the memory model and all main interfaces. There are lot of a code to support the modularity and other logical constraints of the memory model. But the memory model cannot be used without an implementation of memory containers and algorithms.

Any code related to an implementation is placed to appropriate child namespace of the *Implementation* namespace. Some algorithms or structural containers have several variants of an implementation. A user of the memory model is able to select each variant by providing the proper factory to the constructor of the *ModularMemoryModelFactories*.

The user should consider important fact that some combination of algorithms and containers may not work together. For example the merge algorithm which expects an incremental call stack may fail if a structure uses a parallel implementation. Each algorithm and structure has to specify requirements for containers and other algorithms. If the user won't respect these requirements then the memory model won't work properly but may raise a runtime exception.

A version of the memory model used in this text contains several variants of implementation which were designed to illustrate main ideas of this thesis. The definition of each variant can be found in the class *ModularMemoryModelVariants*.

Copy implementation

A copy implementation contains memory structure and algorithms based on the original copy memory model. The implementation follows all basic principles from the third chapter with no extra optimization. This variant is very inefficient and is used only to illustrate the difference between the original and the new memory model.

A code is located in namespaces called *CopyAlgorithms* and *CopyStructure*. There are implementations of all interfaces defined within the memory model.

The whole structural storage is in class *SnapshotStructureContainer*. This class is derived from an *AbstractSnapshotStructure* which implements both structural interfaces *IReadOnlySnapshotStructure* and *IWriteableSnapshotStructure*. The structure container contains all required inner containers and implements all operations to provide the correct functionality of the structural storage with an incremental variant of a memory stack and no change tracker. All inner containers can be replaced using the structural factories so the *SnapshotStructureContainer* class can be reused in a more efficient variant. The structure is accessed using a proxy instance of class *CopySnapshotStructureProxy*.

The namespace *CopyStructure* contains implementation of all structural containers defined within interfaces. All containers implement the behavior described in the previous chapter but each copy request creates the deep copy of the whole container. Please visit documentation of the source code to get the complete list of all containers (all containers starts with prefix *Copy*).

The value storage is implemented by a class *SnapshotDataAssociativeContainer* which implements all requested interfaces. The data container uses the factories to create

an internal associative container. A proxy object is implemented within a class *CopySnapshotDataProxy*.

The whole implementation of the copy memory algorithms can be found within a namespace *CopyAlgorithms*. The algorithm classes are located in child namespaces *MemoryPhase* and *InfoPhase*. The other namespaces contains supporting classes to provide requested operation. A namespace *MemoryWorkers* contains worker and data classes for all algorithms. An implementation of some operation is usually located in some worker class. The algorithm instance creates new worker and use its public interface to pass all parameters and to invoke a computation. A namespace *IndexCollectors* contains an implementation of the collecting process for read and assign algorithms.

An algorithm instance is usually just a proxy with entry points of all operations.

Each worker class can implement more universal operation which may be called from several places. For example the merge algorithm contains several variants of the merge operation. The algorithm has to provide different functionality but the merge process itself is not much different.

Please, visit documentation or source codes to get more information about the copy implementation of memory algorithms.

Lazy algorithms variant

The lazy algorithms implementation improves original copy implementation by adding lazy behavior to memory containers and commit algorithm. Implementation can be found in namespaces *LazyAlgorithms* and *LazyStructure*. All inner containers are the copy implementations from the previous variant.

The lazy algorithms variant uses the same *SnapshotStructureContainer* and *SnapshotDataAssociativeContainer* containers as the copy implementation. The lazy behavior is added by the new implementation of proxy classes *LazySnapshotStructureProxy* and *LazySnapshotDataProxy*.

The variant contains new implementation of the commit and the memory algorithms. Other algorithms use the copy implementation from previous variant. The lazy commit is optimized to not to provide full commit all the time but only if snapshot contains several transactions and memory containers was modified. This behavior has to be supported in

the memory algorithm which has to provide a simplification when a new memory entry is created.

Lazy containers variant

The lazy containers variant is very similar to the previous one. It contains the same algorithms a proxy object. The difference is that all inner containers were re-implemented to add lazy behavior to their internal storages. New implementations are injected to both structural containers using the memory factories.

Tracking variant

The tracking variant is a new variant which implements all main optimizations described in the third chapter. The whole implementation is in namespaces *TrackingAlgorithms* and *TrackingStructure*.

Both memory containers are re-implemented to use a change tracker and a parallel call stack. The change tracker is modified when any update operation tries to modify some memory location of the snapshot. Implementation of memory containers is located in classes *TrackingSnapshotStructureContainer* and *TrackingSnapshotDataAssociative-Container*. By default all inner containers use lazy implementations but they can be changed in factories as usual. Implementation of proxy objects is located in classes *TrackingSnapshotStructureProxy* and *TrackingSnapshotDataProxy*.

The tracking variant re-implements the merge, the commit and the assign algorithms. The new algorithms uses whole potential of the change tracker to reduce time consumed on processing unchanged locations. Motivation and logic of both algorithms is fully described in the chapters 3.3.5 Merge and 3.3.6 Commit. Please visit the source code and the documentation to see details of the tracking implementation.

Note that the copy merge algorithm doesn't work with the tracking memory storages. The original merge expects that a structure contains stack contexts for numbers from zero to a call level. This is no longer happening in parallel call stack.

The new implementation of an assign algorithm doesn't need to create deep copy of assigned data to a new temporary variable. The collecting process is designed to collect all target locations but not to do any change within the structure. New locations and implicit objects are created after collecting process is finished so input memory entry remains unbroken until the assign is performed.

The new assign also fixes a bug of the original algorithm. The copy assign implementation runs the same collecting process for the memory and the info phases. Later was discovered that this approach is wrong because the same collecting process doesn't propagate all possible values from unknown locations. The new assign stores information about source location within a special storage in the snapshot instance – a public property *AssignInfo*. A side effect of this approach is that assign in the second phase doesn't need to perform the collecting process at all.

Differential variant

The differential variant is the same as the tracking variant but it changes used associative container. The previous variant uses lazy implementation of the inner associative containers implemented by class *LazyDictionaryAssociativeContainer.* It uses the simple laziness as other containers. Each instance contains an inner associative container (a C# *Dictionary* class) which can be accessed by the public interface.

The Differential variant uses *DifferentialDictionaryAssociativeContainer*. The differential implementation contains more advanced lazy approach. The copied instance doesn't create a full copy of an inner container but holds list of all differences to the previous states. The read operation has to go back and try to find the first occurrence of a requested item. This approach is able to significantly reduce memory complexity of the whole memory model. A time complexity of read operation is bigger but can be optimized. If the data are merged together when sequence is too big then the average time complexity of the sequence of reads and writes is even smaller than before.

Improved laziness is fully described in chapter 3.2.5.1 Data sharing.

4.3 Starting the analysis

There are several ways to invoke a computation of the Weverca analyzer.

4.3.1 Console application

The first of all there is a console application *Weverca.exe* written for the first release of the Weverca analyzer. The *Weverca.exe* is a simple tool which allows a user to select an analyzed file and choose a memory model. The application prints statistics of the analysis and list of reported warnings to the standard output.

```
C:\Weverca\analyzer\console>Weverca.exe -sa -mm ModularMM C:\Weverca\test_prog
Using Modular copy memory model

File path: C:\Weverca\test_programs\salesperson.php

First phase of the analysis completed in: 5 seconds

Warnings
Total number of warnings: 53
Number of first-phase analysis warnings: 3
Number of taint analysis warnings: 50

Analysis warnings:
--------------------------------------------------------------------------------
File: C:\Weverca\test_programs\salesperson.php
--------------------------------------------------------------------------------
Warning at line 2 char 1: Wrong type of argument number 2 in function ini_set,
Called from:  -> C:\Weverca\test_programs\salesperson.php at position (2,1)-(2

Warning at line 259 char 40: Division by zero
Called from:

Warning at line 392 char 9: Wrong type of argument number 2 in function usort,
Called from:  -> C:\Weverca\test_programs\salesperson.php at position (392,9)-


Security warnings:
--------------------------------------------------------------------------------
File: C:\Weverca\test_programs\salesperson.php
--------------------------------------------------------------------------------
Warning at line 69 char 96: Null value goes into browser
Called from:
```

Image 4:6 An example output of the Weverca.exe

The source code of the console application is located in the project *Weverca* within the main solution of the Weverca analyzer. A deployed version is also part of the attached CD in a folder *analyzer/console*.

Computation can be invoked by the command:

```
Weverca.exe -sa -mm ModularMM filename
```

4.3.2 GUI application

A GUI application is an alternative, more user friendly way to start the computation of the Weverca analyzer. A WPF application was written to support the development of the modular copy memory model. However it is a fully independent part of the Weverca which is able to replace the old console application.

The source code can be located in the main solution in project *Weverca.App*. A deployed version is on the attached CD in a folder *analyzer/gui*. The application is started by *Weverca.app.exe*.

After opening an application a user is prompted by a *Start New Analysis* dialog window. The dialog will ask the user to select source file and allow him to choose properties of the analysis. The user is able to select the type of the second phase of the analysis or leave it deactivated. He is also able to choose one of the supported variant of the modular memory model or set a memory limit for the analysis (the analysis is terminated if the limit is reached).

The last option is to turn on a benchmark of the memory model. If the option is enabled then the memory model starts to collect time and memory data about transactions and algorithms. The number of repetitions defines how many times the analysis should be repeated to get more significant results. Note that this option slows down the analysis and should be enabled only if the user is focused to the performance of the memory model.

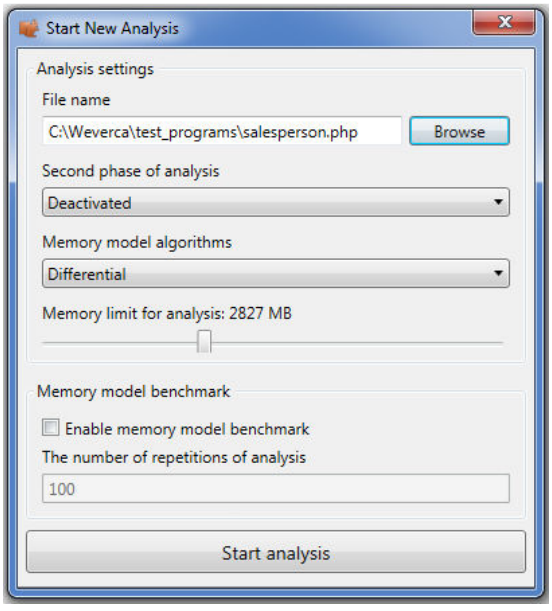


Image 4:7 The Start New Analysis dialog of the Weverca GUI application

When a *Start analysis* button is clicked then the dialog is closed and replaced by the main window with the running analysis. The window consists of two parts: at the left is a panel with an *Abort* button and actualized statistics about running (or just finished) analysis. The central part of the window consists of several tabs which contain an output of the analyzer.

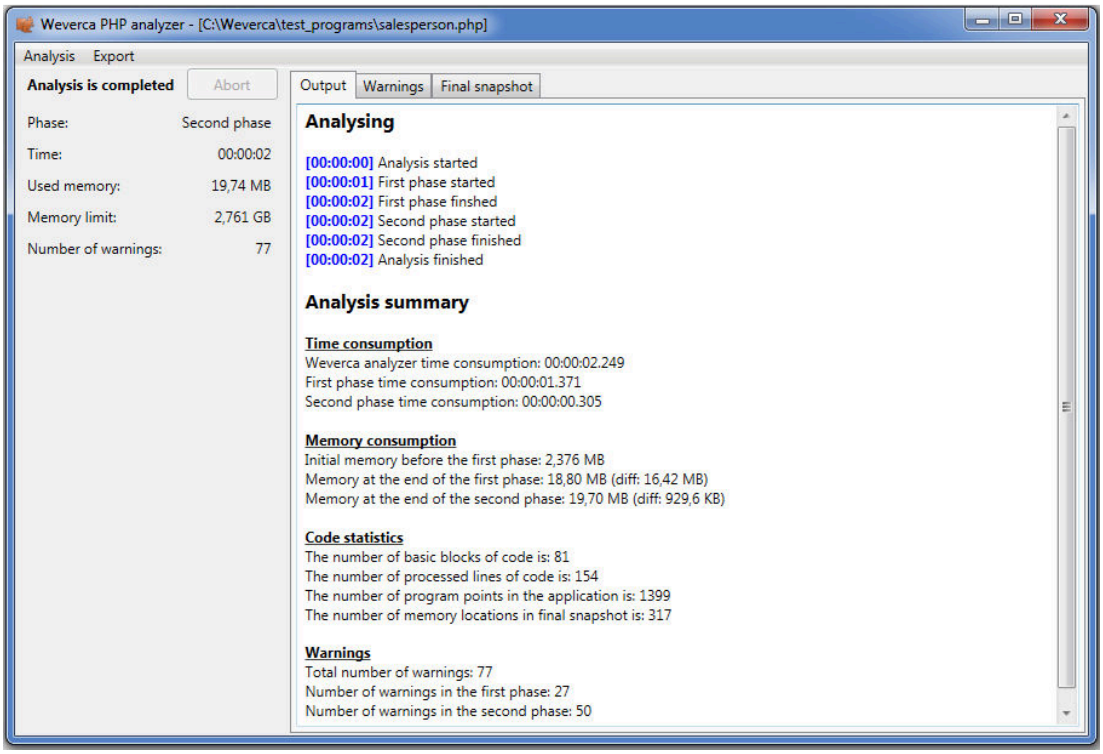


Image 4:8 Main window of the Weverca GUI application

An *Output* tab contains logs and summary of the finished analysis. When the analysis crashes then the *Output* tab will contain full stack trace of the exception. Other two tabs are shown when the analysis is over. A *Warning* tab contains a full list of analysis warnings. A final snapshot shows a complete dump of the memory at the last program point.

Menu items will allow the user to operate the analysis (abort, repeat, start new) or to export results of the last analysis. An *Export analysis dump* menu item will produce single file with all warnings found by the analysis.

An *Export benchmark statistics* menu item is available after the end of a benchmarked analysis. The menu item will export gathered statistics into four CSV files:

- *alg-tot-time* – contains aggregated times of all algorithm types for each benchmark iteration (how much time of the analysis was consumed by each algorithm)

- *benchmark-stats* – contains overall statistics for each benchmark iteration (time, memory, number of transaction …)
- *trans-benchmark* – contains statistics for each transaction in each iteration (ID, time, memory …)
- *trans-mem-med* – contains an average memory consumption for each transaction across all iterations (a memory value is given by a median of transactions with given ID in all iterations)

These statistics was used to compare a performance of variants of the memory model in the chapter 5 Evaluation of memory model algorithms.

Note that the GUI application doesn't support all options of the Weverca analyzer. Its main purpose was to support a functionality needed for the creation of this text. But it is a good starting point for the future development of the final application.

4.3.3 Web application

The first release of the Weverca analyzer was introduced a web application written in an ASP.NET. The application is able to perform full analysis on a PHP source code given by a user.

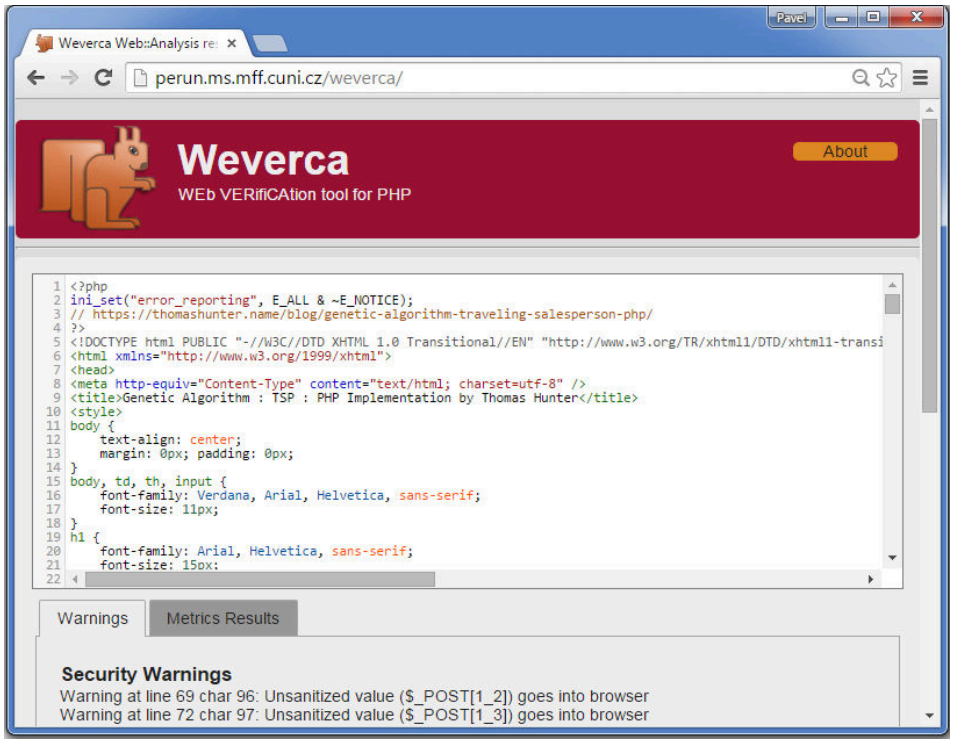


Image 4:9 Weverca web application

The main advantage of the web application is that anyone is able to perform the remote analysis of a custom PHP source. A user is not forced to install and run whole system on his computer but he is able to use resources of the remote server. A public deploy of the web application is the best way to bring more users to the weverca analyzer.

The disadvantage of the current version is that it is not able to perform analysis of the full PHP with includes. The user is limited to post whole PHP code by a single text box in the web application.

The web application is currently deployed (state at the 28th of July 2015):

```
http://perun.ms.mff.cuni.cz/weverca/
```

5 Evaluation of memory model algorithms

The modular copy memory model contains several variants of implementation as described in the fourth chapter. Each variant changes some part of the memory model to introduce some main optimization and its impact to the memory model and the analysis. The best way to evaluate performance of optimizations is to run analysis of the same PHP code for each variant.

There are five distinct variants of implementation: *Copy*, *Lazy algorithms*, *Lazy containers*, *Tracking* and *Differential*. The difference between variants was described in the fourth chapter but the main purpose of each variant is following:

- *Copy* – original implementation
- *Lazy algorithms* – added basic laziness to reduce unnecessary copying during the extend and lazy variant of a commit algorithm
- *Lazy containers* – added lazy behavior to inner containers
- *Tracking* – added a change tracker and a parallel call stack with new tracking merge and commit algorithms
- *Differential* – added differential associative container to tracking algorithms

The following chapters will compare performance of each variant using benchmark tests in the Weverca application. The application is able to collect several performance statistics to compare distinct variants. Each benchmark test will collect and compare: *total time of analysis*, *total memory usage of analysis* and *time consumed by each algorithm*.

A benchmark of each variant runs several iterations of the same analysis to reduce statistical errors caused by testing environment (cache, processor planning, etc.). The final performance statistics are computed as medians of collected data.

There were two testing environments to perform the tests:

- Primary: Notebook Lenovo G570, Intel Core i5-2430 2.4GHz, 4 GB RAM
- Secondary: Intel Core i7-2620M 2.7GHz, 12GB RAM

5.1 Heap sort

The first analyzed code is an OOP implementation of Heap sort algorithm. Full code can be found on the attached CD in section *test_programs/heapsort.php*.

An original code was found on phpcoderblog [7] in an article about sorting algorithms. The heap sort is not a full page example but contains all common techniques used in the PHP which gives a complex view on the analysis and the memory model. The analysis has to process 47 lines of code in 409 program points and produce no warning.

A benchmark was provided by 100 runs of the first phase analysis which should give enough information to compare all implementation variants.

A summary of benchmark stats can be found in a Table 5:1. The table contains medians of the number of transactions, analysis time and memory for each implementation variant. The time and the memory difference columns show percentage improvement between current and *Copy* or a previous tested variant.

	Transactions	Time [ms]	Memory [MB]	Time difference		Memory difference	
				Copy	Previous	Copy	Previous
Copy	1842	911	12.12	-	-	-	-
Lazy algorithms	1996	288	4.71	68%	68%	61%	61%
Lazy containers	1996	278	4.53	69%	3%	63%	4%
Tracking	2000	112	2.57	88%	60%	79%	43%
Differential	2000	117	1.80	87%	-4%	85%	30%

Table 5:1 Heap sort: benchmark results

The first thing you may notice is varying number of transactions (how many times the analysis called *Start Transaction* on a snapshot instance). The difference is caused by distinct implementations of a commit algorithm and its usage of laziness. The new commit doesn't try to find a difference when Start Transaction was called for the first time. This may lead to such a state that an analysis tries process some program point again even if it is not necessary. This is not big deal for the analyzer and even time and memory results show that this approach is correct.

The values in time and memory column are very optimistic. All implementations are more than three times faster than the original variant. Even the used memory at the end of the analysis is at least half of the original value. The best variants are both implementations

of tracking algorithms. The tracking implementation is able to finish in 112 ms which is about 60% faster than the basic lazy variant and saves additional 43% of memory.

Very disappointing are the results of the lazy container variant. The results are slightly better than those from lazy algorithms but 3% of a time and 4% of a memory savings are close to a statistical error. It is a pity because usage of lazy containers promised an interesting improvement, but it looks like the particular version of inner containers doesn't matter.

On the other hand a combination of tracking algorithms and differential associative containers gives very interesting numbers. Usage of differential containers is able to save another 30% of used memory without any significant delay of the analysis.

	Read	Assign	Extend	Commit	Merge
Copy	5	38	97	513	111
Lazy algorithms	4	56	3	73	115
Lazy containers	4	43	2	74	119
Tracking	4	37	2	11	13
Differential	4	31	1	14	19

Table 5:2 Heap sort: algorithm times in milliseconds

The Table 5:2 and the Chart 5:1 show a distribution of time which was spent by each memory algorithm. The worst of all were the extend and especially the commit algorithm which time complexity was reduced by 97%.

Usage of basic laziness had a great impact to the extend and commit operations but slowed down the assign algorithm. Slowing down the assign algorithm may seem strange but it is logical consequence of adding laziness. A snapshot is no longer copied in the extend algorithm but right before the first update which is most likely within the assign algorithm. Note that some portion of copying was moved to other operations which are not a part of any algorithm. Time complexity of the assign algorithm was returned to previous values in tracking implementation when new implementation stopped to copy assigned values to temporary variables.

Another great improvement was achieved by using a change tracker in the new implementations of commit and merge. Note that this code is not complex enough to show the impact of a shared functions supported by tracking merge.

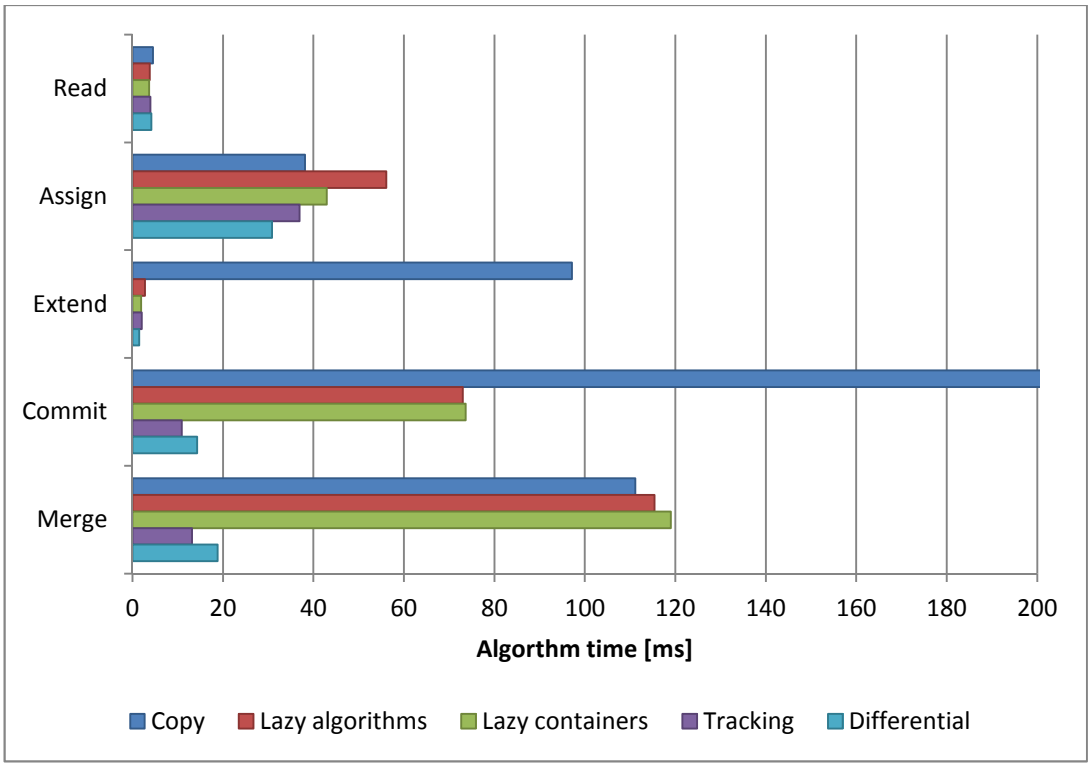


Chart 5:1 Heap sort: algorithm times

5.2 Travelling salesperson

The second benchmark test was running on a PHP implementation of a genetic Traveling salesperson algorithm found on Thomas Hunter's blog [8]. The code is located on the attached CD: *test_programs/salesperson.php*.

The single file web page contains a form to specify visited cities. Submitting the form invokes computation of the algorithm from specified parameters. The weverca analyzer has to process 154 lines of code in 1399 program points.

Each benchmark provided 100 runs of analysis of the first phase.

	Transactions	Time [s]	Memory [MB]	Time difference		Memory difference	
				Copy	Previous	Copy	Previous
Copy	11562	6.85	43.89	-	-	-	-
Lazy algorithms	11188	2.09	15.67	69%	69%	64%	64%
Lazy containers	11188	1.99	14.71	71%	5%	66%	6%
Tracking	12160	1.04	9.60	85%	48%	78%	35%
Differential	12160	1.12	7.44	84%	-7%	83%	23%

Table 5:3 Travelling salesperson: benchmark results

The code of travelling salesman is more complex than previously analyzed heap sort. Single analysis needed more transactions, time and memory to analyze the whole program but there are the same trends as in previous case. The number of transactions is increasing with the used implementation of the commit algorithm. The value of time and memory is reduced by applying laziness and a change tracker. Small changes within time and memory difference columns might be explained by different character of the analyzed code and anomalies within the test environment.

	Read	Assign	Extend	Commit	Merge
Copy	193	172	704	4046	737
Lazy algorithms	198	244	7	649	732
Lazy containers	171	217	7	642	716
Tracking	15	164	8	149	446
Differential	16	165	8	187	513

Table 5:4 Travelling salesperson: algorithm times in milliseconds

Table 5:4 contains time results of each main algorithm. The distribution of values is very similar to the previous case and doesn't need any further explanation.

The main difference between the heapsort and the salesperson code is that salesperson uses a page form to pass a user input into the genetic algorithm. The usage of user input allows for revealing the full potential of the weverca analyzer – its ability to spot runtime and security risks.

The first phase of analysis found 3 runtime and 24 security warnings. The second phase identified another 50 tainted errors.

The Division by zero warning was caused by passing ZERO size of population by the user form. This warning illustrates a perfect usage of the weverca analyzer. The programmer now knows about a possibility of unexpected behavior and may add another test of input values to prevent it. Analysis has also found a wrong usage of two PHP functions when the code tries to pass other type of a parameter than expected. Note that these may be false positives because a program may use deprecated variants of functions.

All 24 security warnings and first 48 of taint warnings were caused by passing POST parameters back to the page form. This is a common issue known as *cross site scripting* (XSS). The attacker is not able to harm a web application itself but there is a possibility that he can prepare a fake request and send it to another user. Consider this fragment of code:

```
<input value="<?=$_GET['val']?>" />
```

Executing following URI will cause that visitor will receive the following code and see an injected message box. Note that this is a harmless example but real attacker can use any vulnerability of visitor's browser to get direct access to his computer.

```
http://page.com?val=%22/%3E%3Cscript%3Ealert%28%22test%22%29;%3C/script%3E%3Ci%
20a=%22
<input value=""/><script>alert("test");</script><i"" />
```

The solution is very easy: all GET and POST variables printed to the browser have to be sanitized by standard PHP function *htmlspecialchars*:

```
<input value="<?=htmlspecialchars($_GET['val'])?>" />
```

The Weverca found all 24 XSS vulnerabilities in the salesperson source code (24 first phase security warnings and 48 second phase analysis warnings). Note that all vulnerabilities are reported three times. It is another challenge for further development to optimize a warning output to group warnings caused by the same cause.

The last two taint warnings were caused by a possible flow of an uninitialized value into the browser.

5.3 School web page

The third analyzed code is a small redaction system written for a basic school Bobnice by author of this thesis. The page contains all typical parts of a redaction system as a connection to database, creation of a view and an administration part. The code also illustrates a typical future usage of weverca analyzer – verification of relatively small systems created by a programmer in order to learn the PHP language. The danger of such systems is that an unskilled programmer may not know typical vulnerabilities and how to protect the page against them. The code can be found on attached CD in: *test_programs/zsbobnice/*.

The code is split into several files included into single entry point *index.php*. The benchmark did not analyze the whole page but just variant of the page which handles showing and creating articles (together with the whole initialization and all other common parts). Another part of the page (gallery, administration) can be analyzed by modification line 30 of index.php.

```
include_once "_php/$_page[script].php";      all variants of page
require_once "_php/article.php";             will select only articles handler
```

The analyzer has to process 783 lines of PHP code in 13811 program points. The benchmark consisted of 10 iterations of a first phase of analysis.

	Transactions	Time [s]	Memory [MB]	Time difference		Memory difference	
				Copy	Previous	Copy	Previous
Copy	36904	53.32	1072.81	-	-	-	-
Lazy algorithms	36666	7.62	270.74	86%	86%	75%	75%
Lazy containers	36666	7.01	233.63	87%	8%	78%	14%
Tracking	37136	2.89	116.46	95%	59%	89%	50%
Differential	37136	2.78	73.31	95%	4%	93%	37%

Table 5:5 School page: benchmark results

The Table 5:5 contains summarization of benchmark statistics. The table contains all main trends spotted in both previous tests. The total speedup of analysis from nearly a minute to couple of seconds is very good. The new implementations also achieved very impressive memory savings – 1GB of original copy algorithms was reduced below 100 MB.

The analysis has also reported 203 warnings – 5 runtime, 53 security from the first phase and 140 taint warnings from the second phase. Most of the security and taint warnings were caused by the *cross site scripting* and *SQL injection* possibilities

The principle of SQL injection attack is very similar to XSS. The attacker tries to use an insecure page form or get parameter to pass a custom SQL command to target database. This vulnerability may cause that attacker to see forbidden data or even modify the state of the database. For example an attacker is able to grant himself an administrator rights to achieve full control over the web application or just drop some table in the database thus making the page inaccessible.

The defense against SQL injection is very simple. The programmer just has to add escape characters into the user input which goes to database. Sanitization can be provided by PHP built-in function *addslashes*. Note that sanitizing GET and POST parameters is not enough. The attacker has full control over all data sent to the server. He is able to build a custom request to fake any header parameter – for example the user agent string or the value of any page cookie.

5.4 NOCC webmail client

The last tested PHP application was the NOCC [6] webmail client which provides access to IMAP and POP3 accounts. NOCC is one of the most complex PHP applications which can be analyzed by the current version of the weverca analyzer with just few modifications. NOCC contains several unsupported PHP constructs which have to be removed before the analysis can be run. A modified version of NOCC has removed exception handling and selected fix action to analyze the part of the system responsible for reading e-mails.

The code can be found on attached CD in: *test_programs/nocc-1.9.4/*.

The input file for analysis is *action.php*. Variable *$action* is set to *aff_mail*. The system contains 2142 lines of PHP code represented as 19428 program points. The following table contains results of a single benchmark run.

	Transactions	Time [s]	Memory [MB]	Time diference		MemoryDiference	
				Copy	Previous	Copy	Previous
Copy	-	-	-	-	-	-	-
Lazy algorithms	70158	144.71	1312.70	-	-	-	-
Lazy containers	70158	137.98	1240.67	-	5%	-	5%
Tracking	67650	16.46	432.48	-	88%	-	65%
Differential	67650	9.28	146.54	-	44%	-	66%

Table 5:6 NOCC webmail client: partial benchmark results

The Table 5:6 contains a comparison of analysis using the lazy and tracking variants. There are no results for copy implementation because the copy ran out of memory. The benchmark was terminated after 11 minutes of analyzing when memory limit 2.5 GB was exceeded. The analyzer reported 6 out of 27 warnings before termination.

But even these partial results are very interesting – speedup from 2 minutes to 10 seconds and more than 1GB saved memory between basic laziness and differential container. These savings are a great success of the tracking merge and commit algorithms. Especially the merge algorithm can finally show the power of shared functions – this can be noticed from a lower number of transactions.

The total effect of optimization can be seen in the Table 5:7 with statistics of benchmark run on the second test environment. The same NOCC code was analyzed by x64 version of weverca analyzer on secondary test machine with 12 GB RAM.

	Transactions	Time	Memory	Time diference		MemoryDiference	
				Copy	Previous	Copy	Previous
Copy	70256	368,44	5929,08	-	-	-	-
Lazy algorithms	70158	75,12	2097,02	80%	80%	65%	65%
Lazy containers	70158	71,09	1999,94	81%	5%	66%	5%
Tracking	67650	10,25	635,60	97%	86%	89%	68%
Differential	67650	7,16	237,96	98%	30%	96%	63%

Table 5:7 NOCC webmail client: benchmark results

The tracking variants reported 52 warnings (7 runtime, 20 security and 25 tainted) whereas the lazy variants reported 56 warnings (additional 1 security and 3 tainted).

An additional security warning was located in the *action.php* because of an unsanitized ANY value which went to the browser. This is a false positive warning which was generated because an old implementation of shared functions mixed contexts of distinct stack levels. The new tracking implementation was able to prevent propagation of the value out of some shared function into the global context.

Three extra tainted warnings were located in file *html/header.php*. The reason of these false positives was very similar as above.

6 Conclusion

The main goal of this thesis was to create a new version of a memory model for the Weverca analyzer. The requested memory model should improve possibilities of the original copy memory model which was a part of the first release of the Weverca. The original memory model had several functional and design issues which blocked further usage of the whole analyzer.

The new memory model took the best ideas from the original one and integrated them into a new modular framework. A modularity of the new memory model helped to perform tests and benchmarks to identify bottle necks and to apply several optimizations.

All problems of the original memory model were fully exposed when the first optimization was run for the first time. Even the basic improvement (lazy extend algorithm) had significant effect to the performance of whole analyzer. All optimizations together were able to speed up an analysis of whole systems from several minutes to a couple of seconds and to reduce memory consumption from gigabytes to hundreds of megabytes.

All numbers introduced in the Chapter 5 Evaluation of memory model algorithms show that the optimization process was successful. The main goal of this thesis was accomplished.

But despite the obtained numbers are great there is still space for more optimizations. Very promising is potential of differential containers which are currently used only as main containers within a structural and value storage. Further research may reveal that differential containers may be successfully used across the whole memory model. An implementation of differential containers is not ideal. The associative containers can be re implemented to use more clever techniques to save another portion of memory and speed up searching the stored element.

A big challenge for the future development is to create a parallel implementation of memory algorithms. Currently all operations within a single algorithm are performed in a single thread. Multithreading may improve the performance of an analyzer by speeding up all main algorithms. A further research is needed but parallelism may be potentially used within the assign, a commit and a merge algorithms.

The future work may also focus on an implementation of some of well known optimization of static analysis. The memory model already supports shared functions but there are more techniques connected with a memory model which might optimize total performance of

analysis. One of them is a mechanism of allocations sites. This is an optimization used to reduce the number of created arrays and object instances. If some array or object is created in a cycle then the analysis will create one special array or object and distribute it across all related locations. The current implementation of the analysis already supports simple allocation sites of objects but more research is needed.

The development of the Weverca shouldn't stay focused only to a memory model. The memory model certainly consumes the biggest portion of memory of all parts of the analyzer. Future research should investigate performance of other parts as well. Especially the taint analysis needs to be cleaned and optimized.

The future of the Weverca analyzer is very promising. The Weverca is still not able to perform a full analysis of the big systems but optimizations of memory model open more possibilities of a future grow.

Bibliography

[1] NIELSON, Flemming, NIELSON Hanne Riis and HANKIN Chris. *Principles of Program Analysis:* Springer, 2005

[2] CLARKE, Justin. *SQL Injection Attacks and Defense, Second Edition*: Syngress, 2012

[3] HAUZAR, David, KOFROŇ, Jan and BAŠTECKÝ, Pavel. *Data-flow Analysis of Programs with Associative Arrays*.
In: Proceedings of the International Workshop on Engineering Safety and Security Systems (ESSS'14), Singapore, EPTCS, 2014

[4] *Weverca project: Web verification for PHP*
[online]. [cit. 28.7.2015].
Available at: http://d3s.mff.cuni.cz/projects/formal_methods/weverca/

[5] *Phalanger project: The PHP Language Compiler for the .NET Framework*.
[online]. [cit. 28.7.2015].
Available at: http://phalanger.codeplex.com

[6] *NOCC: Host your own Webmail Client*.
[online]. [cit. 28.7.2015].
Available at: http://nocc.sourceforge.net/

[7] *PHPCODERBLOG: PHP some sorting algorithms: bubble sort, selection sort, counting sort, quicksort, shellsort, heapsort.*
[online]. [cit. 28.7.2015].
Available at: https://phpcoderblog.wordpress.com/2013/02/26/php-some-sorting-algorithms-bubble-sort-selection-sort-counting-sort-quicksort-shellsort-heapsort/

[8] *Thomas Hunter II: Genetic Algorithm Traveling Salesperson PHP.*
[online]. [cit. 28.7.2015].
Available at : https://thomashunter.name/blog/genetic-algorithm-traveling-salesperson-php/

A. List of Images

Image 2:1 Architecture of the Control flow graph builder

Image 2:2 An architecture of the first phase analysis framework

Image 2:3 An architecture of the second phase analysis framework

Image 3:1 An input and an output state of a program point

Image 3:2 A memory representation of the Virtual References memory model

Image 3:3 A memory representation within the Copy memory model

Image 3:4 An array identification and description

Image 3:5 A linked list with a cyclic reference

Image 3:6 Stack and heap in PHP

Image 3:7 Call levels in a call stack

Image 3:8 Top level branches of a memory tree

Image 3:9 An example of an array stored in a variable

Image 3:10 An Example of simplified memory tree

Image 3:11 Lazy structure

Image 3:12 Partial sharing

Image 3:13 Differential containers

Image 3:14 Mixed local context in stack by call level

Image 3:15 Context separation in parallel call stack

Image 3:16 The Parallel call stack – Indirect recursion starting from several functions

Image 3:17 The main loop of interpreting memory path by the BFS algorithm

Image 3:18 The processing of an index or a field segment

Image 3:19 Resolving locations for the direct path segment when location is a) defined and b) undefined

Image 3:20 Resolving locations for an unknown path

Image 3:21 Flow of The read algorithm

Image 3:22 Flow of the assign algorithm

Image 3:23 Resolving a memory location when the location is undefined

Image 3:24 Implicit array within an index access in the collecting process

Image 3:25 An implicit object within a field access in the collecting process

Image 3:26 Flow of the assign alias algorithm

Image 3:27 The merge of two snapshots

Image 3:28 Flow of the merge algorithm

Image 3:29 Back references of trackers in conditional branches

Image 3:30 Back references of memory trackers at the end of merge process
Image 3:31 The sequence of program points in the shared functions
Image 3:32 Back references of memory trackers at the end of the shared function
Image 4:1 A layer architecture of the snapshot object
Image 4:2 The block diagram of the snapshot structure object
Image 4:3 A block diagram of the Stack context identified by a stack level number
Image 4:4 A block diagram of index container with an associative array of variables
Image 4:5 A block diagram of an index definition object
Image 4:6 An example output of the Weverca.exe
Image 4:7 The Start New Analysis dialog of the Weverca GUI application
Image 4:8 Main window of the Weverca GUI application
Image 4:9 Weverca web application

B. List of Tables

Table 5:1 Heap sort: benchmark results

Table 5:2 Heap sort: algorithm times in milliseconds

Table 5:3 Travelling salesperson: benchmark results

Table 5:4 Travelling salesperson: algorithm times in milliseconds

Table 5:5 School page: benchmark results

Table 5:6 NOCC webmail client: partial benchmark results

Table 5:7 NOCC webmail client: benchmark results

C. Content of attached CD

The folder structure of the attached cd is as follows:

- analyzer — *The deployed versions of the analyzer*
  - console
  - gui
- benchmark_results — *Each subfolder contains the results of some benchmarking test*
  - heapsort
  - nocc-12GB
  - nocc-lenovo
  - salesperson
  - zsbobnice
- test_programs — *List of test programs which can be processed by the analyzer*
  - nocc-1.9.4
  - zsbobnice
  - dijkstra.php
  - fractal.php
  - heapsort.php
  - salesperson.php
  - simple_xss.php
- weverca_source_code — *Folder with a complete source code of the weverca analyzer*
- master_thesis.pdf — *An electronic version of this text*