PHP AiR: Analyzing PHP Systems with Rascal

Mark Hills* and Paul Klint†‡
*East Carolina University
Greenville, North Carolina, United States of America
mhills@cs.ecu.edu
†Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
‡INRIA Lille Nord Europe
Lille, France
Paul.Klint@cwi.nl

***Abstract*—PHP is currently one of the most popular programming languages, widely used in both the open source community and in industry to build large web-focused applications and application frameworks. To provide a solid framework for working with large PHP systems in areas such as evaluating how language features are used, studying how PHP systems evolve, program analysis for refactoring and security validation, and software metrics, we have developed PHP AiR, a framework for PHP Analysis in Rascal. Here we briefly describe features available in PHP AiR, integration with the Eclipse PHP Development Tools, and usage scenarios in program analysis, metrics, and empirical software engineering.**

I. Introduction

PHP,[1] invented by Rasmus Lerdorf in 1994, is a dynamic object-oriented language focused on server-side application development. It is now one of the most popular languages, as of October 2013 ranking 5th on the TIOBE programming community index,[2] used by 78.8 percent of all websites whose server-side language can be determined,[3] and ranking as the 4th most popular language on GitHub by repositories created in 2013 (as of October 27).[4] This popularity has led to the creation of a number of large, widely-used open source applications and application frameworks, including WordPress,[5] Joomla,[6] Drupal,[7] MediaWiki,[8] Symfony,[9] Magento,[10] and CodeIgniter,[11] and has made it a popular choice for developers creating new web applications and frameworks.

The availability of such large, open-source systems provides an ideal ecosystem for empirical software engineering research. PHP is also an important target for program analysis research. Most PHP applications are web-based, giving an urgency to analyses focused on detecting potential security errors. At the same time, the dynamic nature of the language (e.g., duck typing, reflection, dynamic inclusion of source files, runtime construction and evaluation of code), as well as its use on larger and larger systems, increases the importance of analyses targeted at program understanding, automated refactoring, and programmer tool support, all areas where PHP currently lags behind languages such as Java.

To enable research in these areas for PHP systems, we are developing PHP AiR[12], an environment for PHP Analysis in Rascal. We first introduce PHP AiR in Section II, giving a high-level overview of the current tool, including a brief discussion of usage scenarios, some current research being performed using PHP AiR, and integration with Eclipse. In Section III we then provide details on four specific use cases for PHP AiR, along with initial results in each: feature usage in PHP applications; static analysis to resolve dynamic includes in PHP programs; hybrid static/dynamic analysis to convert uses of dynamic features to static variants of these features; and changes in dynamic feature usage in WordPress as the software has evolved.

II. PHP Analysis in Rascal

The PHP AiR framework is made up of a number of tools and reusable libraries built to support research in empirical software engineering, software evolution, program analysis, and software metrics for systems built using PHP. Figure 1 shows an overview of PHP AiR. PHP AiR itself is built using a combination of PHP, Java, and the Rascal meta-programming language [1]; the language used for any given step is indicated in each box. Boxes with rounded corners represent tools provided by the system or tasks created by users (e.g., custom queries over a PHP system), while boxes with square corners represent artefacts, such as the original PHP system, abstract syntax trees (ASTs) for PHP files, or results of running PHP AiR, such as tables, files with analysis results, visualizations, or other documents.

Starting with a PHP system, made up of one or more PHP source files, PHP AiR creates ASTs for each of the files. These ASTs are the basic structure used by PHP AiR to

[1] http://www.php.net
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[3] http://w3techs.com/technologies/details/pl-php/all/all
[4] The query for this is based on http://adambard.com/blog/top-github-languages-for-2013-so-far/.
[5] http://wordpress.org/
[6] http://www.joomla.org/
[7] http://drupal.org/
[8] http://www.mediawiki.org/wiki/MediaWiki
[9] http://symfony.com/
[10] http://www.magentocommerce.com/
[11] http://ellislab.com/codeigniter
[12] https://github.com/cwi-swat/php-analysis

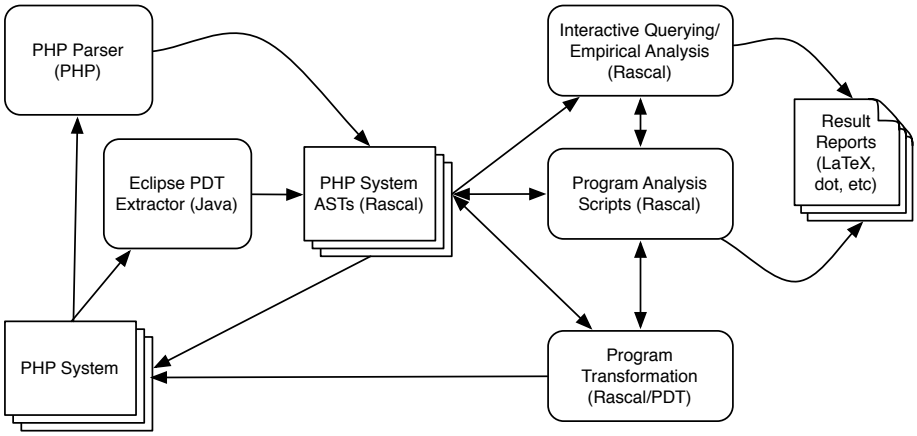

Fig. 1. High-Level Overview: PHP AiR.

reason about PHP code, and are used along with information computed using Rascal (e.g., call graphs, library information extracted from the PHP documentation) as well as limited information (e.g., lines of code information for the files in the systems, minimum required PHP version and release date of each system) provided by outside tools in CSV files and read in using Rascal's resources feature [2]. These ASTs are organized into *systems*, where a single system represents all the files and associated ASTs for a given product and version (e.g., MediaWiki 1.19.1, WordPress 3.6).

PHP AiR makes use of two different parsers, depending on whether the desired usage scenario requires speed or precision. The first is an external parser for PHP, also written in PHP, that is based on the parser used inside PHP itself. This parser can be used to parse individual files, but is especially well suited to batch processing large numbers of PHP files, e.g., all the source files across all releases of WordPress. Parsing returns either a single AST, a single system, or multiple systems, depending on the input. The second is based on the parser used by the PHP Development Tools (PDT) in Eclipse. When files are parsed by the PDT, an internal document-object model for each file, similar to an AST, is created. This DOM is then traversed by the Eclipse PDT Extractor to extract ASTs identical to those given by the first parser for each file. The PDT Extractor is targeted at interactive use within Eclipse, and requires all source files to be part of an Eclipse project open in the same workspace. Either individual files or individual systems (one for each project) are extracted.

Using these ASTs and systems, PHP AiR provides a number of Rascal functions and data types aimed at code querying, empirical software analysis, software evolution, program analysis, and program transformation. These are packaged as Rascal libraries, allowing them to be reused in other Rascal code to support specific analysis tasks or empirical investigations. Since systems and ASTs are represented as Rascal types, PHP AiR can look at individual ASTs, a single system, or even multiple systems at the same time, for instance to compare how features are used in different products (e.g., MediaWiki versus WordPress) or to see how the use of specific features has changed across different versions of the same product (e.g., usage of `eval` across different releases of WordPress).

Some detailed examples of existing tools that have been created using PHP AiR are given in Section III; other examples include a taint analysis for detecting dangerous uses of unchecked user-provided strings in PHP library calls, a refactoring from hand-coded HTML to uses of template libraries, a similar refactoring from hand-coded SQL calls to uses of database libraries, and multiple projects to extract various metrics from PHP source code. In many cases the reported results have been directly generated by PHP AiR and saved in formats such as CSV files, GraphViz dot diagrams, and even LaTeX-formatted tables and figures. Examples of the latter are shown in this paper in Table I and Table II.

Along with extracting information from PHP systems and modifying the ASTs, transformed PHP code can also be generated. This can be done in one of two ways. First, PHP AiR ASTs can be pretty-printed to generate their concrete representations; the limitation is that this does not keep whitespace, including comments, so this is best in situations where specific code needs to be injected into specific locations of an existing file, or where a new file is being created from scratch. Second, PHP AiR interfaces with the PDT to allow existing statements or expressions to be replaced with new statements or expressions, allowing arbitrary code to be injected into specific locations in PHP files. This is limited to existing files which must be part of a PHP project in Eclipse.

III. Current Uses of PHP AiR

While Section II described general usage of PHP AiR and briefly mentioned several projects, in this section we describe some of our ongoing work using PHP AiR in more detail.

A. Empirical Analysis of PHP Feature Usage

To support our work on program analysis for PHP, we performed an empirical study of PHP language feature usage [3] across a corpus of 19 large open-source web applications and

TABLE I
USAGE OF INVOCATION FUNCTIONS.

System	Files			CUF	CUFA	CUM	CUMA	Gini
	Total	Inv	Inc					
CakePHP	640	28	34	9	30	0	0	0.17
CodeIgniter	147	6	9	5	3	0	0	0.17
DoctrineORM	501	10	10	3	9	0	0	0.15
Drupal	268	24	40	10	30	0	0	0.30
Gallery	505	20	22	28	23	0	0	0.46
Joomla	1,481	25	30	26	25	0	0	0.41
Kohana	432	8	8	5	7	0	0	0.17
MediaWiki	1,480	89	250	69	80	0	0	0.29
Moodle	5,367	87	1,073	95	69	0	1	0.31
osCommerce	529	2	2	2	0	1	0	0.17
PEAR	74	10	10	20	10	0	0	0.45
phpBB	269	11	44	9	13	0	0	0.31
phpMyAdmin	341	5	5	0	6	0	0	0.13
SilverStripe	514	27	27	32	21	0	0	0.31
Smarty	126	7	8	4	8	0	0	0.29
SquirrelMail	276	3	36	2	2	0	0	0.17
Symfony	2,137	60	62	39	34	0	0	0.15
WordPress	387	39	82	44	50	0	0	0.41
ZendFramework	4,342	92	92	73	86	0	0	0.34

application frameworks consisting of $3,370,219$ source lines of code, including such well-known systems as WordPress, MediaWiki, Drupal, Symfony, and the Zend Framework. This work set out to answer the following questions:

- How large are real PHP programs?
- How often are the various features of the PHP language used in these programs?
- Which features need to be defined precisely to faithfully capture the core of PHP as it is used in practice, and which little-used features could be modeled with less precision?
- Where and how often are some of the harder to analyze language features, such as dynamic file inclusion, evaluation of arbitrary code given in strings at runtime, and dynamic invocation of functions (where the function name is given as a variable) used in existing PHP code?
- Are these dynamic features spread evenly through the code, or do they tend to cluster in specific files?
- Are uses of dynamic features truly dynamic, or is it possible to capture patterns that can be leveraged in analysis tools?

The reported results [3] were all computed using PHP AiR, with the computation scripted to ensure the results are reproducible, and with the various tables and figures given in the paper generated directly using Rascal. An example of such a table, showing the use of dynamic invocation functions, is shown in Table I. Each system in the corpus is given on one row; the Files columns show the total number of files in the system (Total), the number of files that include a dynamic invocation (Inv), and this number plus the number of files that indirectly include a dynamic invocation through file inclusion (Inc). The next four columns show the number of uses of invocation functions `call_user_func`,[13] `call_user_func_array`,[14] `call_user_method`,[15] and `call_user_method_array`.[16]

[13]http://php.net/manual/en/function.call-user-func.php
[14]http://php.net/manual/en/function.call-user-func-array.php
[15]http://php.net/manual/en/function.call-user-method.php
[16]http://www.php.net/manual/en/function.call-user-method-array.php

The final column gives the Gini coefficient, which measures the distribution of occurrences of dynamic invocations in files among those files with at least one dynamic invocation. In general, the Gini, ranging from 0.0 to 1.0, is used to compute inequality, with lower numbers meaning that the quantity being measured is more evenly distributed between the members of the population (here, files) and higher numbers meaning that occurrences tend to cluster, with some files having very few and some files containing many occurrences. Files with no occurrences are not included in the computation; most files have no occurrences, so their inclusion would drive up the Gini coefficient unnecessarily (if many files do not use the feature, this would be seen as extreme inequality, driving the number towards 1.0) and make it meaningless.

B. Resolving PHP Dynamic Includes

One specific example of a problematic dynamic feature identified in the work discussed above is PHP's file inclusion mechanism, which allows the name of a file to be included to be computed at runtime using an arbitrary expression. Inclusion then occurs at runtime as well: some parts of the file are brought in at runtime as new top-level definitions (e.g., classes and functions) and other parts of the file run directly in the context of the including file. This provides an obvious challenge for code maintenance and analysis—it may not be possible to statically know the code that will actually be executed, blocking possible analysis tasks and transformations.

In prior work [3], we showed that many of these dynamic includes are actually static in practice, with string building expressions used in these cases not to provide for multiple possible include files but, instead, to simplify the process of defining the path to the intended file and to provide for configurability. Taking advantage of this, we have developed an includes resolution analysis that identifies these dynamic includes and, where possible, links them to the specific file in the system that will be included. Resolving these dynamic includes improves static analysis and code understanding, ensuring that analysis and transformation tasks have access to the actual code that will be executed at runtime and making it possible to apply more powerful analysis techniques [4].

Table II shows the results of this analysis on the same corpus used above: the software product is in the first column, while the next three columns show the total number of includes, the number of dynamic includes, and the number of dynamic includes that can be resolved into unique files at runtime. The next two columns show the number of files in the system and the number with unresolved dynamic includes. The final two columns give the percent of resolved includes and the distribution of unresolved includes in those files containing at least one (see the discussion of the Gini above for details). Overall, of the $7,962$ dynamic includes in the corpus, we are able to resolve $6,523$ (81.92%) to unique include files.

C. Transforming Dynamic Features to Static Features

Recent work in JavaScript [5]–[7] and Ruby [8], [9] has focused on using a combination of static and dynamic

TABLE II
PHP DYNAMIC INCLUDES IN THE ORIGINAL CORPUS.

Product	Includes			Files		% Resolved	Gini
	Total	Dynamic	Resolved	Total	Unresolved		
CakePHP	124	120	67	640	25	55.83	0.44
CodeIgniter	69	69	28	147	20	40.58	0.44
DoctrineORM	56	54	36	501	14	66.67	0.19
Drupal	172	171	130	268	16	76.02	0.42
Gallery	44	39	25	505	10	64.10	0.26
Joomla	354	352	193	1,481	127	54.83	0.18
Kohana	52	48	4	432	18	8.33	0.55
MediaWiki	554	493	461	1,480	25	93.51	0.18
Moodle	7,744	4,291	3,625	5,367	382	84.48	0.37
osCommerce	683	539	497	529	22	92.21	0.28
PEAR	211	11	0	74	9	0.00	0.14
phpBB	404	404	326	269	40	80.69	0.37
phpMyAdmin	819	52	15	341	27	28.85	0.23
SilverStripe	373	56	27	514	10	48.21	0.34
Smarty	38	36	25	126	7	69.44	0.29
SquirrelMail	426	422	406	276	13	96.21	0.14
Symfony	96	95	41	2,137	40	43.16	0.22
WordPress	589	360	332	387	17	92.22	0.32
ZendFramework	12,829	350	285	4,342	42	81.43	0.29

techniques to identify usage patterns of dynamic features and replace them with static variants of these features. We are currently working on a similar transformation for PHP. Using PHP AiR, points of interest are identified in the program source. The website is then run using a headless web testing tool, which generates a number of execution traces showing values that reach these points of interest, such as the strings that contain code to be evaluated or functions to be invoked dynamically. Using the values, uses of dynamic features are then transformed using PHP AiR into equivalent static features (e.g., a dynamic function invocation is transformed into direct calls of the functions that could be invoked, an eval is transformed into the various blocks of code that could be evaluated), with an optional fall-back path for the dynamic case if this is still reachable. Although this is ongoing work, our early results show that this could be a valid approach both for exploring actual runtime behaviors and for specializing the code used to support plug-ins such as those found in WordPress.

D. Changes in Dynamic Feature Usage in WordPress

While the above use cases focused on specific systems or on comparisons across different products, PHP AiR can also be used to compare different versions of the same product. We are currently exploring how WordPress has evolved, based on all available public releases of the system. We are looking specifically at how the use of dynamic features has changed during this time. For instance, although version 3.6 of WordPress does not use eval, the first standard release included 6 uses, with the peak over the lifetime of WordPress being 8 in a handful of versions (out of around 75 versions total that we are analyzing). With eval and other dynamic features, we are looking both at the patterns of use over time and investigating (by looking at the source code and commit messages) why uses of these features have been added and removed. Our goal is to categorize uses of these features in various systems, identifying patterns that can be exploited to allow us to transform uses of these dynamic features into uses of other features that are safer and easier to analyze.

REFERENCES

[1] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM'09*. IEEE, 2009, pp. 168–177.

[2] M. Hills, P. Klint, and J. J. Vinju, "Meta-language Support for Type-Safe Access to External Resources," in *Proceedings of SLE'12*, ser. LNCS, vol. 7745. Springer, 2012, pp. 372–391.

[3] ——, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of ISSTA'13*. ACM, 2013, pp. 325–335.

[4] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu, "The HipHop Compiler for PHP," in *Proceedings of OOPSLA'12*. ACM, 2012, pp. 575–586.

[5] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proceedings of PLDI'10*. ACM, 2010, pp. 1–12.

[6] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications," in *Proceedings of ECOOP'11*, ser. LNCS, vol. 6813. Springer, 2011, pp. 52–78.

[7] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval Begone!: Semi-Automated Removal of Eval from JavaScript Programs," in *Proceedings of OOPSLA'12*. ACM, 2012, pp. 607–620.

[8] M. Furr, J. An, and J. S. Foster, "Profile-Guided Static Typing for Dynamic Scripting Languages," in *Proceedings of OOPSLA'09*. ACM, 2009, pp. 283–300.

[9] M. Furr, J. An, J. S. Foster, and M. W. Hicks, "Static Type Inference for Ruby," in *Proceedings of SAC'09*. ACM, 2009, pp. 1859–1866.