Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications

Hung Viet Nguyen
ECE Department
Iowa State University

Christian Kästner
School of Computer Science
Carnegie Mellon University

Tien N. Nguyen
ECE Department
Iowa State University

ABSTRACT

In plugin-based systems, plugin conflicts may occur when two or more plugins interfere with one another, changing their expected behaviors. It is highly challenging to detect plugin conflicts due to the exponential explosion of the combinations of plugins (i.e., configurations). In this paper, we address the challenge of executing a test case over many configurations. Leveraging the fact that many executions of a test are similar, our variability-aware execution runs common code once. Only when encountering values that are different depending on specific configurations will the execution split to run for each of them. To evaluate the scalability of variability-aware execution on a large real-world setting, we built a prototype PHP interpreter called *Varex* and ran it on the popular WordPress blogging Web application. The results show that while plugin interactions exist, there is a significant amount of sharing that allows variability-aware execution to scale to 2^{50} configurations within seven minutes of running time. During our study, with Varex, we were able to detect two plugin conflicts: one was recently reported on WordPress forum and another one was not previously discovered.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation, Measurement

Keywords

Variability-aware Execution, Testing, Configurable Code, Plugin-based Web Applications, Software Product Lines

1. INTRODUCTION

A plugin is a software component that contributes functionality and adds features to an existing software application. Plugin-based applications offer a variety of benefits such as allowing third-party developers to extend an application and supporting easy addition and configuration of new features for different needs. For these reasons, plugin-based systems are becoming increasingly popular. Examples include the Mozilla add-ons framework for the Firefox browser [5], the add-in extension mechanisms used in Microsoft Office [4], the plugin architecture of the Eclipse platform [2], and the WordPress Web blogging software [12].

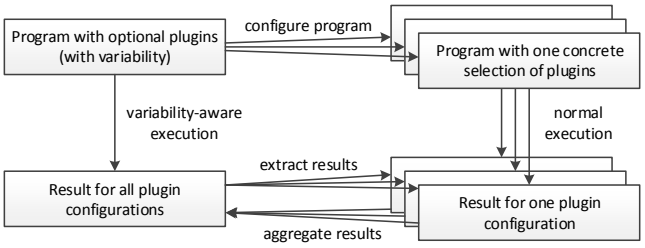

Figure 1: Variability-aware vs. brute-force execution

Plugin conflicts in such plugin-based systems are not uncommon. Plugin conflicts arise in the cases where one plugin interferes with another plugin's behavior when they are used together, even though both work as expected in isolation (also known as the feature-interaction problem [19, 42]). Importantly, one plugin may accidentally violate another plugin's assumptions or override or bypass its behavior by modifying shared state. In fact, the developers of WordPress have stated that plugin incompatibility is the top reason that people feel unwilling to upgrade to WordPress' latest version [10].

Plugin conflicts are notoriously hard to detect upfront. Plugin behavior is rarely fully or even formally specified, making the approaches based on formal methods or requirements engineering used for detecting feature interactions in telecommunication systems [19, 42] rather hard to apply on large-scale, plugin-based software like WordPress. The developers could write test cases and execute them on individual plugin configurations to detect conflicts. However, *executing* the test cases for plugins is challenging due to the *combinatorial explosion* of the number of plugin configurations, which in practice is often compensated only by manual and ad-hoc approaches, as reported for the Eclipse project [32].

In this paper, we tackle the challenge of executing a test case *exhaustively* over all configurations of a software product (i.e., all combinations of a set of plugins). The key observation that allows us to scale such execution despite the exponential explosion is that many executions of a test are similar. With a variability-aware interpreter, we execute common code only once and, only when encountering a configuration option, execute multiple branches with the respective configurations, after which we continue to execute the rest again only once if possible. Conceptually, we run a test case in all configurations without configuring the program first. The result is equivalent to configuring the program in all configurations, running the configurations in isolation, and aggregating the results (Figure 1).

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

(a) WordPress' source code (simplified)

```
1  // Initialize WordPress
2  include('load.php'); ...
3  // Load plugins
4  $plugins = wp_get_active_plugins();
5  foreach ($plugins as $plugin)
6      include($plugin); ...
7  // Print scripts
8  foreach ($wp_scripts as $script)
9      echo "<script type='text/javascript' src='$script' />"; ...
10 // Print content
11 $content = wp_get_content();
12 foreach ($wp_content_filters as $func)
13     $content = call_user_func_array($func, $content);
14 echo $content;
```

(b) The Smiley plugin

```
1  wp_enqueue_script('jquery', 'jquery-1.7.js'); ...
2  function convert_smileys($content) {
3      return $content.replace(':]', get_smiley(':]'));
4  }
5  $wp_content_filters[] = 'convert_smileys';
```

(c) The Weather plugin

```
1  wp_enqueue_script('jquery', 'jquery-2.0.js'); ...
2  function insert_weather_widget($content) {
3      return $content.replace('[:weather:]', get_weather_widget());
4  }
5  $wp_content_filters[] = 'insert_weather_widget';
```

Figure 2: Simplified WordPress and two plugins

The efficiency of variability-aware interpretation depends on how variability is used in the application. If plugins have local effects and do not interact, it can be very efficient. A few interactions slow down the execution since we need to compute with alternative values for some variables, but still much sharing remains so that far fewer alternatives need to be explored than the brute-force approach. Thus, the feasibility of variability-aware interpretation is mainly an empirical question (prior work, though promising, has only investigated small or artificial cases; see related work section).

We evaluate the feasibility of variability-aware interpretation in a large-scale practical setting: executing a test case over WordPress with optional plugins. We build Varex, a prototype variability-aware interpreter for PHP to evaluate how variability from plugin activation affects test execution in WordPress and how plugins interact. In our empirical study on real-world plugins, we found that, nearly 28% of executed statements and 89% of variables' values are shared among all plugin configurations. We found plugins in WordPress are in fact mostly orthogonal or interact mostly in disciplined ways, rendering our approach practical. Due to low interaction among plugins, Varex was able to cover 2^{50} possible configurations within seven minutes. With these promising results, we hope that in the long run, variability-aware execution will establish a scalable testing and analysis mechanism for configurable systems of different kinds.

The key contributions of this paper include: (1) a variability-aware execution technique for running PHP plugin-based Web applications, (2) Varex, a prototype variability-aware PHP interpreter, and (3) an empirical study showing the scalability of variability-aware execution for testing configurable systems in a large real-world scenario.

2. MOTIVATING EXAMPLE

In this section, by means of an example, we illustrate potential conflicts in a plugin-based system, the challenges in detecting such plugin conflicts, and the opportunities of sharing that we can exploit for variability-aware execution.

(a) Output with Weather activated and Smiley deactivated

```
1  <script type='text/javascript' src='jquery-2.0.js' /> ...
2  Weather forecast: <div>...Temperature: 76°F...</div>
```

(b) Output with both Weather and Smiley activated

```
1  <script type='text/javascript' src='jquery-1.7.js' /> ...
2  Weather forecast: [:weather<img src='smile.gif' />
```

Figure 3: Conflict between Weather and Smiley

We selected the popular open-source, PHP-based blog software *WordPress* [12] as the subject of our study, since it represents a typical plugin architecture with over 25,000 available plugins, and is broadly used on over 60 million websites (from basic blogs to large-scale portals and enterprise websites). It is well-known for being prone to plugin conflicts [6]. It exhibits common plugin conflict characteristics as studied in other plugin systems [32].

WordPress is implemented as a classic framework, to which plugins can contribute additional functionality by registering callback functions to various events (following the observer and strategy design patterns) or modifying shared global state. For illustration, we show a strongly simplified core of WordPress and two plugins in Figure 2. After initializing its own running environment (line 2), WordPress retrieves plugins in alphabetical order from a database (line 4) and initializes them by calling the PHP include function for every plugin (lines 5-6). Plugins can register required JavaScript libraries through shared states ($wp_scripts), which are printed on lines 8-9 as HTML <script> tags. Finally, WordPress receives a blog post from the database and prints it after applying filters that plugins may have registered (lines 11-14).

Plugin Conflicts. Plugin conflicts can arise when two or more plugins interfere with one another's behavior. While some conflicts such as conflicting function names lead directly to crashes, others manifest themselves silently and cause the page to display incorrectly. In our simplified WordPress example, in Figure 2, the plugins Smiley and Weather conflict: The Smiley plugin converts smiley codes within a blog post (e.g., ':]') into images, whereas the Weather plugin injects a weather widget in the post; in addition, both plugins use different versions of the same jQuery library. Both plugins work well in isolation, but produce unexpected results when combined as shown in Figure 3. As the first conflict, plugin Smiley replaces string ':]' in the '[:weather:]' tag before plugin Weather can act, resulting in unexpected output. Plugin Smiley also includes a version of the jQuery library older than needed by plugin Weather, resulting in runtime JavaScript errors. In this example, the plugin that is initialized first affects the behavior of the plugin that follows; however, in general, a plugin can also invalidate the effects of the plugins that run before it.

Worse, platforms such as WordPress operate under an open-world assumption, where third parties can contribute plugins and not all plugins and contributors may be known. That is, plugin conflicts may occur late when a user actually combines two (or more) plugins, possibly from different sources, in the same environment. It is unrealistic for developers to anticipate all interactions with other plugins.

Challenges in Testing for Plugin Conflicts. Plugin conflicts are notoriously hard to detect upfront. They typically arise from incompatible assumptions, inconsistent requirements, conflicting goals, overlapping preconditions (e.g., both plugins replace overlapping strings as in Figure 3), or conflicting postconditions (e.g., both expect different library versions) [42].

Beyond opportunistic and ad-hoc testing of individual configurations [32, 43], developers could write simple test cases validating

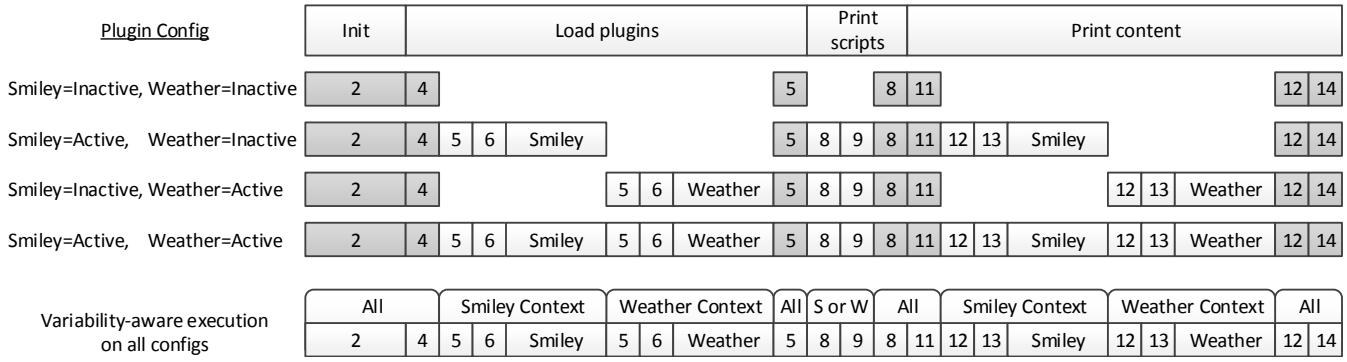

Figure 5: Shared statements among the execution of different plugin configurations

(a) Test case for the Weather plugin

```
1  function testWeather()
2      output = runWebPage('index.php') // Execute the Web page
3      // Perform assertions on the output
4      assertContains(output.getElementByXPath('/html/body/div[1]'),
           'Temperature');
5      assertEqualsOrNewer(output.get ... ByXPath('string(// script [1]/
           src)'), 'jquery-2.0.js')
```

(b) Expected test results

```
Assertion failed at testWeather, L4 if Smiley∧Weather.
Assertion failed at testWeather, L5 if Smiley∧Weather.
```

Figure 4: Example test case for the Weather plugin

their assumptions. How to write such test cases is well established and common practice [9]. As an illustration, in Figure 4a we show a test case for the Weather plugin querying the main page and checking whether the '[:weather:]' tag has been replaced correctly and whether the jQuery library has an up-to-date version. Such a test can identify a plugin conflict when executed on a configuration with both the Weather and the Smiley plugins.

To detect conflicts with test cases early, however, the test cases need to be *executed* on individual plugin configurations. Every WordPress user can install and execute any combination of plugins in their system. As more and more plugins are added, the number of possible combinations of plugins grows *exponentially*. With 20 plugins, the number already runs into the millions; with more plugins brute force quickly becomes entirely infeasible. Therefore, executing the tests on many configurations in the development phase is expensive. Sampling strategies, e.g. pairwise sampling [43, 24], can help to reduce the test effort, but are intrinsically incomplete. Shipping test cases with the plugins [32] delays test execution until a user actually combines plugins, delays the detection and resolution of conflicts, and shifts responsibilities to the users.

Approach Overview. We tackle the challenge of executing a test case exhaustively over all configurations of a software product (i.e., all combinations of a closed set of plugins from a repository). The key observation that allows us to scale such execution despite the exponential explosion is that many executions of a test are very similar. For example, independent of the configurations, the entire framework initialization (line 2 in Figure 2 for our motivating example) and the retrieval of the posts (line 11) is always the same. Plugins typically have local effects and rarely interact; some plugins may not even affect the executed test case at all.

To highlight sharing in our example, we illustrate the shared statements among the execution in four configurations in Figure 5. For each configuration, a sequence of boxes shows the line numbers of the statements in the main WordPress program (Figure 2) that are executed. The statements in the plugins' source code are grouped together and denoted by Smiley and Weather, respectively. The boxes across different configurations are aligned vertically to reveal the shared code among them. Several statements (marked with a dark background) are shared among all four configurations. The amount of shared code in the actual WordPress system would be even higher (e.g., the initialization step would include far more statements than just the statement on line 2).

Leveraging such code sharing on the execution paths for different plugin configurations, instead of running each configuration separately, we develop a *variability-aware interpreter* that executes the code for all configurations in a single run. The variability-aware interpreter executes common code only once and when encountering a configuration option, it executes multiple branches with the respective configurations. Our interpreter is able to operate on a data representation capable of capturing alternative values of variables during executing all configurations. As an illustration, the single execution trace of the variability-aware execution is shown at the bottom part of Figure 5. For each statement, the interpreter keeps track of a *variability context*, which describes which part of the configuration space it is executing. For instance, statements 8 and 9 are executed under the context that either Smiley or Weather is activated, while statements 2 and 4 are executed for all configurations.

Our variability-aware interpreter executes all configurations once and yields results equivalent to brute-force runs (Figure 1). Importantly, since variables can have alternative values in different configurations, the test case may fail in only parts of the configuration space, pinpointing plugin conflicts to the problematic configurations. For example, the src property of tag <script> in the example's output is 'jquery-1.7.js' if Smiley is activated and 'jquery-2.0.js' otherwise. Thus, the assertion on line 5 of Figure 4a would fail if and only if both Smiley and Weather were activated (Figure 4b).

3. VARIABILITY-AWARE EXECUTION

Informally, a regular execution for one program configuration can be viewed as a sequence of computations on data. In contrast, variability-aware execution is a sequence of computations on *multi-value data*, whose values may differ between configurations. We introduce **Varex**, a PHP variability-aware interpreter that performs computations on multi-value data. Let us present its data representation and computations.

3.1 Variability-Aware Data Representation

Let us first formulate important concepts. Then, we explain Varex's data representation and how its variability-aware computation works.

```
1  $foo = PluginConfigOption('Foo'); // PluginConfigOption('X') yields
2  $bar = PluginConfigOption('Bar'); // True iff X is activated
3  if ($foo)
4      $content = 'Running [Foo]';
5  else
6      $content = 'Welcome';
7  $content = str_replace ('[Foo]', 'Plugin Foo', $content);
8  $status = '';
9  if ($foo || $bar)
10     $status = ' (Plugins on)';
11 echo $content . $status;
```

Figure 6: A PHP program with variability

DEFINITION 1 (CONFIGURATION OPTION). *A* **(plugin) configuration option** *for a plugin p is a Boolean variable that assumes* True *if p is activated and* False *otherwise.*

DEFINITION 2 (CONFIGURATION). *A* **configuration** *is a specific assignment of Boolean values to configuration options that specifies which plugins are activated.*

DEFINITION 3 (PLUGIN-BASED APPLICATION). *A* **plugin-based application** *is a family of programs defined by the activation of plugins. A configuration corresponds to a specific program.*

DEFINITION 4 (CONFIGURATION SPACE). *A* **configuration space** *is the set of all possible configurations. A configuration space for n plugins has* 2^n *configurations.*

DEFINITION 5 (VARIABILITY CONTEXT). *A* **variability context** *is a subset of the configuration space. We describe it with a propositional formula over configuration options, which yields* True *for all configurations belonging to the variability context and* False *otherwise. The formula representing a variability context is* satisfiable *iff the variability context contains at least one configuration.*

For example, variability context True contains all configurations, variability context False contains none, and the context $\mathsf{Foo} \wedge \neg \mathsf{Bar}$ describes the possibly very large set of configurations in which plugin Foo is activated and plugin Bar is not activated (while all other plugins can either be activated or not). The concepts can be extended for non-Boolean configuration options with a finite domain.

In designing our variability-aware data representation, we aim to achieve two goals: (1) the representation should be able to represent a concrete value for every variable in every configuration, and (2) the representation should be so compact that the same value in multiple configurations are represented once instead of several times. Thus, we partition the configuration space with variability contexts. For all configurations inside a context, a variable shares the same value.

To illustrate the sharing, we show a simple PHP program with two plugins in Figure 6. (The new example is simpler than the previous one for readability, while still containing important PHP constructs that will be used to explain our ideas.) As shown in Figure 7, instead of keeping the values of $content for all four configurations, we partition the configuration space into two parts (with contexts Foo and $\neg$Foo) and maintain one distinct value for each context. Specifically, we manage data variability with an abstract data type called **MultiValue**, with two concrete subtypes:

1. **ConcreteValue**: A ConcreteValue represents a concrete PHP value that does not depend on any configurations.
2. **Choice**: A Choice models two alternative multi-values depending on a variability context. Choice(ϕ, x, y) denotes that the value is x for all configurations in the variability context ϕ and y otherwise.

Plugin Config	Value of $content before line 7, Figure 6
(1) Foo=False, Bar=False	'Welcome'
(2) Foo=True, Bar=False	'Running [Foo]'
(3) Foo=False, Bar=True	'Welcome'
(4) Foo=True, Bar=True	'Running [Foo]'
Variability-aware execution on all configs	Choice(Foo, 'Running [Foo]', 'Welcome')

Figure 7: Variability-aware data representation

To hierarchically partition the configuration space, choices can be nested. Our representation roughly follows the Choice calculus [28]. In our example, we represent the two string values as Choice(Foo, ConcreteValue('Running [Foo]'), ConcreteValue('Welcome')) (last row in Figure 7). For readability, we omit the ConcreteValue construct in the rest of the paper.

3.2 Variability-Aware Computation

Varex's computation is realized with the following ideas:

1. Shared data. Varex aims to represent differences between configurations compactly. It uses choices only for those variables whose values actually differ. A choice between two equivalent values can be simplified (Choice(ϕ, x, x) $\rightarrow$ x), and a choice of similar objects can be compacted as one object with common values of the objects' fields being factored out.

2. Shared execution. Varex performs execution on shared code just once and *splits* the current variability context only when variability occurs in values (e.g., a read variable has multiple values) or in the control flow (e.g., the condition of an if statement evaluates to different values). Varex shares execution as long as possible (called *late splitting*, see Section 3.2.2). After a split, the next statements are executed in restricted variability contexts (similar to path conditions in symbolic execution). Then, the results from the computations in those variability contexts are aggregated again into one compact value and used for the next shared computations (*early merging*). The goal is to execute each statement under the largest possible variability context.

3.2.1 Store and Load Operations on Shared Data

As introduced above, the value of variables may depend on the variability context. By representing sharing with MultiValue, we can manage variables' values compactly via a map from each variable's name to its corresponding MultiValue (line 1, Figure 8). During execution, Varex maintains a current variability context ϕ, which can change as the execution explores different parts of the configuration space. The two key operations, storing and loading variables' values, are performed under a variability context ϕ as follows.

First, storing a value v to a variable (lines 2-6, Figure 8) means that the variable will have its assigned value v in context ϕ and retain its previous value in other configurations. Therefore, its new value is represented by a Choice of the assigned value and its existing value (line 4). If a variable does not have a previous value, we use a special UNSET symbol as in a regular PHP interpreter to indicate that the variable is uninitialized in some configurations. Helper function createChoice performs some simplifications of the representation (lines 11-22). Second, loading a variable's value in a given context ϕ (lines 7-10) is done by finding values in the variable's MultiValue that satisfy context ϕ via function extract (lines 24-32). It recursively extracts values from the two branches of a Choice by eliminating branches unsatisfiable to ϕ.

Compacting compound structures. In PHP Web applications, the use of compound data structures such as objects and arrays is common. While such data structures can be large, differences often lie

```
1  map = Map[String, MultiValue]; // Maps variables' names to values
2  void storeVariable(String name, MultiValue value, Context φ) {
3      oldValue = map.contains(name) ? map[name] : UNSET;
4      newValue = createChoice(φ, value, oldValue);
5      map[name] = newValue;
6  }
7  MultiValue loadVariable(String name, Context φ) {
8      value = map[name];
9      return extract(value, φ);
10 }
11 MultiValue createChoice(Context φ, MultiValue x, MultiValue y) {
12     if (!satisfiable(¬φ)) return x;
13     if (!satisfiable(φ)) return y;
14     if (x == y) return x;
15     if (x is Object && y is Object) // Compact Choice of objects
16         obj = new Object();
17         for (f in (x.fields ∪ y.fields)) do
18             obj.addField(f, createChoice(φ, x[f], y[f]));
19         return obj;
20     ... // Arrays are handled similarly to objects
21     else return Choice(φ, x, y);
22 }
23
24 MultiValue extract(MultiValue v, Context φ) {
25     if (v is Choice(ω, x, y))
26         x' = extract(x, φ ∧ ω);
27         y' = extract(y, φ ∧ ¬ω);
28         if (!satisfiable(φ ∧ ¬ω)) return x';
29         else if (!satisfiable(φ ∧ ω)) return y';
30         else return createChoice(ω, x', y');
31     else return v;
32 }
```

Figure 8: Storing and loading variables

only in individual fields. Thus, instead of representing choices between objects, we compactly represent a single object with choices in its fields. For an object's fields, we use the same storage representation used for variables. When storing a value, Varex performs this compression recursively in the helper function createChoice (lines 15-19, Figure 8). The result is a compact representation of the original objects, as illustrated below.

```
Original value: Choice(Foo, Object(x => 1, y => 2), Object(x =>
    1, y => 3))
Compacted value: Object(x => 1, y => Choice(Foo, 2, 3))
```

Since PHP arrays are also associative maps from keys to values, similarly to the handling of objects, Varex compacts a Choice of arrays as an array of Choice elements.

3.2.2 Splitting and Merging of Variability Contexts

In a regular program, the result of a computation is always a concrete value. For example, the evaluation of the condition at an if statement returns either True or False, deciding which branch to be run next. In a variability-aware execution, however, the result can be a *multi-value*, i.e., its concrete value may depend on the configuration. Thus, a variability-aware computation may need to be executed on data under specific variability contexts (e.g., both branches of an if statement can be run under different contexts in which the condition evaluates to True and False, respectively).

Specifically, when a computation encounters multi-values, the *current variability context is* **split** *into subcontexts and the execution continues in those subcontexts*. The execution is sequential from one context to another. Since the subcontexts can in turn split in subcomputations, splitting can take place multiple times before the original statement is entirely executed and the execution can proceed with the original context (i.e., the subcontexts are *merged* to form the original context). Splitting occurs on (1) a control statement with a multi-value condition, resulting in execution in different branches, or (2) a computation (e.g., expression) on multi-values (e.g., $a+$b),

```
1  MultiValue execute(IfStatement ifStmt, Context φ) {
2      MultiValue condValue = eval(ifStmt.condExpr, φ);
3      Context ω = whenTrue(condValue);
4      if (satisfiable(φ ∧ ω))
5          execute(ifStmt.thenBranch, φ ∧ ω);
6      if (ifStmt.hasElseBranch() and satisfiable (φ ∧ ¬ω))
7          execute(ifStmt.elseBranch, φ ∧ ¬ω);
8  }
9  Context whenTrue(MultiValue v) {
10  match (v)
11    case ConcreteValue(val)⇒ if(val) return True else return False;
12    case Choice(φ,x,y)⇒ return φ∧whenTrue(x) ∨ ¬φ∧whenTrue(y);
13 }
```

Figure 9: Context splitting on a control statement

causing the same computation to be executed multiple times in different variability contexts.

Context splitting on a control statement. Since the value of the condition of a control statement (e.g., if and while) can be a multi-value, Varex first needs to determine the variability context ω in which the value is True (using function whenTrue in Figure 9) and the context in which the value is False ($\neg\omega$). For an if statement, given the current variability context ϕ, Varex then runs the then and else branches in restricted variability contexts $\phi \wedge \omega$ and $\phi \wedge \neg\omega$, respectively (lines 4-7). Note that in an empty context (i.e., if the corresponding formula is not satisfiable), the corresponding branch does not need to be executed. After running both branches, the execution continues with the original context ϕ.

Figure 10 demonstrates the execution with the splitting and merging of variability contexts for our running example of Figure 6. The boxes show snapshots of the variables' values at different points in the execution (unchanged values are abbreviated with '_'). Executed statements (the transitions between snapshots) are annotated with line numbers and their corresponding variability contexts (in brackets). The graph on the right visualizes the splitting and merging of those contexts. The execution is sequential from one context to another, yet variable accesses take effect in only their respective contexts as explained in Section 3.2.1. The splitting at an if statement is illustrated by the first split (from L3[True] to L7[True]).

Other control statements (e.g., while) are handled similarly to an if statement: If the condition returns a multi-value in which the True value exists in some context, the loop will continue in that restricted context. The body of a while may be repeatedly run in increasingly smaller variability contexts until the loop terminates also in the last configuration (i.e., the variability context is empty). Note that the loop bound is concrete but may depend on the configuration.

Context splitting on a computation. If values involved in a computation (e.g., the operands in expressions) are multi-values, Varex will execute the computation multiple times for individual concrete values represented by the multi-values in their corresponding variability contexts. In essence, for unary operations, we perform a *map* over all concrete values of a multi-value. For binary operations, we map over all combinations of concrete values (with their corresponding intersected variability contexts). We sketch our algorithm in Figure 11. Note that we compress the resulting value into a compact one with the createChoice function. Nevertheless, binary operations may lead to a combinatorial explosion of concrete values with small variability contexts in the worst case. Finally, n-ary operations and function calls with multiple arguments can be handled similarly.

Importantly, although any computation with multi-values can always be done by mapping over all alternative values, Varex defers this splitting if the computation itself can support multi-values (*late splitting*). Specifically, for non-native computations such as a call to a user-defined function, the execution continues without splitting

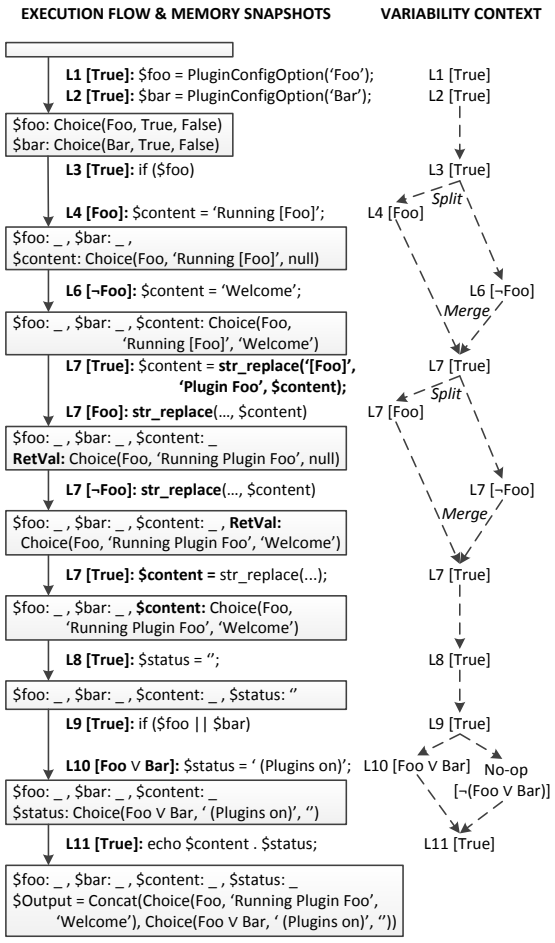

Figure 10: Variability-aware execution for the program in Figure 6

at the call site, since variability can be handled later inside the function's body. For native PHP operations such as addition or logical conjunction, and calls to native PHP functions (e.g., strpos), we either extend the interpreter to enable the native construct itself to handle multi-values or (by default) let the execution split the context using the algorithm in Figure 11. The splitting at a call to a native PHP function (str_replace) is illustrated at the (upper) L7[True] statement in Figure 10.

3.2.3 Evaluation Rules

Let us summarize our evaluation rules for common PHP program constructs in Table 1. The current context ϕ is initialized with the entire configuration space (True).

R1-R6. Variable reading/writing and control statements are handled as explained in Sections 3.2.1 and 3.2.2.

R7. Rule R7 handles control-flow breaking statements (e.g., return, break, continue) and exceptions. Within a block of statements, when such instructions are encountered, a controlFlag variable will get a non-null value, indicating that the remaining statements will not be executed. In the case of an exception, Varex reports the current context ϕ where the exception occurs. Since controlFlag can be a multi-value, we use helper function whenNull (analogous to whenTrue), and continue the execution only in configurations with a null control flag. If a control flag is activated in all configurations (i.e., the remaining variability context is empty), the execution stops for that block of statements.

```
1  // Execute a unary operation on a multi-value
2  MultiValue execute(Op op, MultiValue v, Context φ) {
3      match (v)
4          case ConcreteValue(val):
5              return executeConcrete(op, val, φ);
6          case Choice(ω, x, y):
7              val1 = execute(op, x, φ ∧ ω);
8              val2 = execute(op, y, φ ∧ ¬ω));
9              return createChoice(ω, val1, val2);
10 }
11 // Execute a binary operation on two multi-values
12 MultiValue execute(Op op, MultiValue a, MultiValue b, Context φ){
13     if (a is Choice(α, a1, a2))
14         return createChoice(α, execute(op, a1, b, φ ∧ α),
                                  execute(op, a2, b, φ ∧ ¬α));
15     else if (b is Choice(β, b1, b2))
16         return createChoice(β, execute(op, a, b1, φ ∧ β),
                                  execute(op, a, b2, φ ∧ ¬β));
17     else if (a is ConcreteValue(a') and b is ConcreteValue(b'))
18         return executeConcrete(op, a', b', φ);
19 }
```

Figure 11: Context splitting on a computation

R8-R10. Expressions and native function calls with potential multi-values are handled as explained in Section 3.2.2.

R11-R12. To make string concatenations with possible multi-values at echo/print statements efficient, Varex uses a data type called Concat, which represents a concatenation of string values or multi-values. In R12, $Output is used for the output string. For example, the output value of the code in Figure 6 is shown in the snapshot box after L11 in Figure 10.

R13. Rule R13 supports testing of Web applications using assert statements. Since the value of an asserted expression can be a multi-value, Varex collects all the contexts in which that value evaluates to False and reports those contexts.

3.2.4 Implementation

With the main goal of exploring the feasibility of variability-aware execution in a large-scale practical setting, we extended the open-source full-scale PHP 5 interpreter Quercus, written in Java [7]. For specifying variability contexts and checking satisfiability, we use TypeChef's library for propositional formulas with a JavaBDD backend [34, 3]. We extended Quercus' type system to support multi-values such that concrete values occurring in a regular execution still behave as expected. However, the interpreter now needs to handle operations involving multi-values. Since rewriting all existing operations on regular values into variability-aware operations on multi-values at once is a daunting task, we implemented them incrementally. During execution, we dynamically logged the code locations where operations on multi-values were attempted but not yet supported. We implemented variability-aware alternatives for those operations until no further unsupported operations were logged.

Limitations. Currently, our interpreter implements variability in all operations needed in our evaluation, but not for all of PHP's large API. For instance, we did not implement a variability-aware version of function count (determining the length of an array) yet, because it was never called on arrays of variable length in our system.

Generally, Varex is limited regarding side effects outside the control of the interpreter, e.g., if a plugin writes to a file or makes a state-changing request to a web server. Varex may execute the corresponding code multiple times under different variability contexts, changing the behavior compared to brute-force execution. We did not address this issue yet, because it was not relevant for our experiments. There are many strategies to explore, such as an abstraction layer for a variability-aware file system or a mechanism to avoid joining after potentially uncontrolled side effects [35, 37, 13, 53].

Table 1: Evaluation Rules

PHP Syntax	Evaluation Rule in Satisfiable Context ϕ eval(E, ϕ)
R1. $V = E	storeVariable(V, eval(E, ϕ), ϕ)
R2. $V	loadVariable(V, ϕ)
R3. $V->k = E or $V[k] = E	v=loadVariable(V, ϕ), map=getKeysVals(v) map.storeVariable(k, eval(E, ϕ), ϕ)
R4. $V->k or $V[k]	v=loadVariable(V, ϕ), map=getKeysVals(v) map.loadVariable(k, ϕ)

R5. if (E) S_1
else S_2
	ω = whenTrue(eval(E, ϕ))
	if (sat($\phi \wedge \omega$)) execute(S_1, $\phi \wedge \omega$)
	if (sat($\phi \wedge \neg\omega$)) execute(S_2, $\phi \wedge \neg\omega$)

R6. while (E)
S
	while (true)
	$\phi = \phi \wedge$ whenTrue(eval(E, ϕ))
	if (sat(ϕ)) execute(S, ϕ) else break

R7. {S_1 ... S_n}	for (i = 1 to n) do
	controlFlag = execute(S_i, ϕ)
	if (controlFlag is exception) report ϕ
	$\phi = \phi \wedge$ whenNull(controlFlag)
	if (!sat(ϕ)) break

R8. $\otimes$E	execute($\otimes$, eval(E, ϕ), ϕ)
R9. $E_1 \otimes E_2$	execute($\otimes$, eval(E_1, ϕ), eval(E_2, ϕ), ϕ)
R10. native_func({E_i})	execute(native_func, {eval(E_i, ϕ)})
R11. E_1 . E_2	Concat(eval(E_1, ϕ), eval(E_2, ϕ))
R12. echo E	eval($Output = $Output . E, ϕ)
R13. assert(E)	χ = ¬whenTrue(eval(E, ϕ)), report $\phi \wedge \chi$

Correctness. To ensure that our implementation is correct, we compared Varex's results with those of the execution of individual configurations, following the schema in Figure 1. Specifically, we automated comparing the HTML output and all values in the heap at the end of the execution (except for nondeterministic values, such as the current time). For 10 plugins, we performed this comparison against all 1024 configurations (brute force). For additional plugins, we sampled the configuration space. We executed the comparison for the test described in Section 5 and obtained equivalent results, which gives us confidence in the correctness of Varex.

4. TESTING FRAMEWORK

With its variability-aware execution technique, Varex provides a framework to run a test case in all possible plugin configurations. The process consists of three steps:

1. Initializing optional plugins. In WordPress, the activated plugins are stored in an array in which each value points to an activated plugin's path. To make plugin activation optional, we instrument that code and create an array in which all entries are optional; each of them is guarded by its corresponding configuration option:

```
if (PluginConfigOption('Smiley'))    $plugins [] = 'Smiley_path';
if (PluginConfigOption('Weather'))   $plugins [] = 'Weather_path';
...// Note that PluginConfigOption('P') yields Choice(P,True,False)
```

2. Executing a test on a web page. In general, test cases can be written according to common practice for testing Web applications [9] without any further consideration for variability (Figure 4a). Varex then runs the test case. In contrast to a regular execution in which the output is a concrete string value, Varex's variability-aware execution typically returns a multi-value, representing the output values for all possible plugin configurations.

3. Performing assertions and reporting test results. Assertions in the test case are checked against the (possibly multi-value) output and may throw an exception only in specific variability contexts. Thus, Varex can report that the test succeeds for all configurations or pinpoint failed assertions to specific variability contexts (out of which sample configurations can be generated with a SAT solver).

Table 2: Excerpt of 50 Tested WordPress Plugins

					Contribution	
	Plugin Name	Version	Files	LOC	Chars	Stmts
1	Jetpack	2.2.5	123	50,605	78	2,836
2	Types	1.4.0.1	164	46,071	0	1,719
3	Google Analyticator	6.4.4.3	64	41,287	0	2,614
4	WP Photo Album Plus	5.1.2	56	30,156	3,513	3,655
12	My Calendar	2.1.3	27	12,614	7,940	35,313
25	WP SlimStat	3.3.2	15	4,733	344	4,320
48	R. Simple CAPTCHA	1.6	2	293	0	16
49	WP Facebox	1.2.2	1	109	914	4,171
50	Lazy Load	0.5	1	64	312	844
	Total size of plugins:		1,478	463,374		
	Size of WordPress 3.4.2:		388	183,502		

For example, Varex reports that the two assertions in Figure 4a failed in variability context Smiley $\wedge$ Weather (as shown in Figure 4b), thereby indicating a conflict between Smiley and Weather plugins.

5. EMPIRICAL STUDY

In this case study, we want to assess the feasibility of variability-aware execution in a large real-world scenario. Specifically, we aim at answering the following questions.

(RQ1) Sharing and interaction among plugins. The key idea that allows the variability-aware execution technique to scale is to take advantage of the sharing among plugins. Thus, we study the sharing and interactions among plugins from three different aspects: the output, computations, and values of variables. For each aspect, we ask: How many characters/computations/values are shared among all plugin configurations? How many of them depend on one or more configuration options? How often do plugins interact? In addition, we report all detected plugin conflicts.

(RQ2) Scalability of variability-aware execution. How much time does it take to run Varex in a large configuration space?

Experiment Setup. To address our research questions, we installed the WordPress system with a set of 50 plugins of various domains and sizes. We selected the 40 most popular plugins as listed on WordPress website and 10 plugins that are reported to have had conflicts with some other plugins (to bias our selection slightly toward hard and interacting plugins). Table 2 shows the sizes of the largest and smallest plugins in our set, together with the size of WordPress shown in the last row. The complete list of plugins is available on our website [8]. Next, we created a test case that generated the *home page* of WordPress with a single blog post and initialized optional plugins as described in Section 4. We then ran the test case with Varex and collected data.

5.1 Sharing and Interactions of Plugins - RQ1

5.1.1 Sharing and Interactions Observed in Output

Since the main purpose of a Web application is to generate an HTML output, we first report how plugin interactions manifest in this output. In the multi-value output produced by the execution, we can derive a variability context ϕ for each character, indicating the configurations in which it appears. If the character is produced in variability context True, it is shared by all configurations; if it depends on two or more plugin configuration options appearing in ϕ, it indicates that the character has been produced through an interaction among the corresponding plugins. We counted the distinct configuration options in each character's variability context and reported the aggregated numbers.

Figure 12 shows that around 9,000 characters are shared by all configurations (column 0), and nearly 22,200 characters depend on

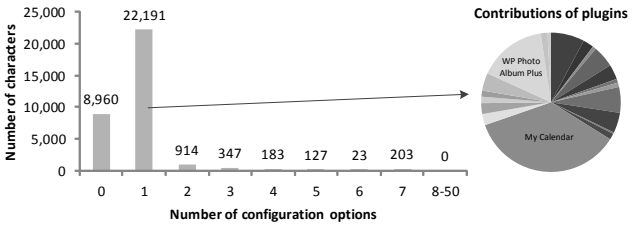

Figure 12: Variability in the program's output

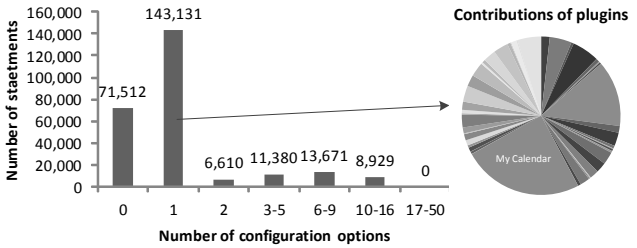

Figure 13: Variability in computations

exactly one configuration option (column 1). That is, 94 % of the output either is common among all configurations or is contributed by plugins independently. The pie chart and Table 2 detail the contribution exclusive to each plugin. It shows that plugins *My Calendar* and *WP Photo Album Plus* contribute the most to the output; they display a calendar widget and a photo slide show in the test web page. Some plugins (29 out of 50 plugins) do not contribute to the main page's output at all.

A few fragments in the output are produced only if multiple plugins are combined, with a maximum of 7 plugins (columns 2-7). We found that most of those fragments are related to interactions in declaring JavaScript libraries, since several different plugins register the same JavaScript libraries with WordPress (as demonstrated in Section 2). For example, plugin *Cardoza* (*CAR*) will print a jQuery script if plugins *WP Facebox* (*FAC*) and *WP Photo Album Plus* (*WPP*) have not already done so. Therefore, the following HTML fragment is displayed under variability context $\mathsf{CAR} \wedge \neg\mathsf{FAC} \wedge \neg\mathsf{WPP}$: <script ...jquery.js?ver=1.7.2'></script>.

5.1.2 Sharing and Interactions in Computations

After studying the output, we are interested in variability of internal computations. We counted each executed statement and analyzed the corresponding variability context and aggregating results as for the output. Figure 13 reports the sharing and interactions among executed statements. As seen, 28 % of the executed statements are shared among all configurations. 56 % of them are specific to one plugin, as further detailed in the pie chart and Table 2. All plugins are executed, with *My Calendar* contributing the most to the execution. Interactions among multiple plugins account for 16% of the executed statements and involve a maximum of 16 plugins. The highest interaction involves plugins accessing the same WordPress *filters* to register callback functions. Another common interaction of plugins occurs in function get_locale of WordPress:

```
L28: function get_locale () { ...
L31:     if ( isset ( $locale ) )
L32:         return apply_filters ( 'locale', $locale );
```

Among others, plugins with IDs *ADV, ALL, BET, DIS,* and *GOO2* attempt to retrieve the locale of the system via get_locale, in which the global variable $locale is set if it was not set earlier. Thus, when plugin *Google Analytics for WordPress* (*GOO2*) calls get_locale, line 32 is executed when *GOO2* is activated, and $locale is set (i.e., one of the other plugins is activated). Thus, the variability context at line 32 is $\mathsf{GOO2} \wedge (\mathsf{ADV} \vee \mathsf{ALL} \vee \mathsf{BET} \vee \mathsf{DIS})$.

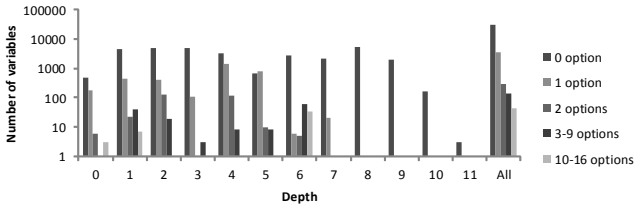

Figure 14: Variability in values (logarithmic scale)

Note that if we executed WordPress in a brute-force fashion for all 2^{50} configurations, we would execute statements in columns 0 and 1 each 2^{50} and 2^{49} times, respectively. In contrast, in a variability-aware execution, those statements are executed only once, reducing execution effort significantly.

We also measured how often a *computation* in a statement is split into two or more subcomputations (due to context splitting). Out of 255,233 executed statements, there are only 3,225 such cases. In 90 % of those cases, the context is split into only two subcontexts. In the following worst case, the context is split into 48 subcontexts:

```
// WordPress-3.4.2/wp-includes/post-template.php
L166: $content = apply_filters ('the_content', $content);
L167: $content = str_replace (']]>', ']]&gt;', $content);
```

Here, the blog-post content (variable $content) is modified by different plugins that registered to contribute to the WordPress the_content filter (line 166). Unless some values can be merged, the number of values of $content doubles with every optional filter, reaching 48 alternative unique values in our case. Subsequently, at line 167, since str_replace is a native function call, Varex splits the current context into 48 subcontexts and executes the call multiple times for concrete values of $content. Conceptually, the combinatorial explosion can be avoided by providing a variability-aware implementation of str_replace that can handle multi-value strings.

5.1.3 Sharing and Interactions in Values

To see plugin interactions in variables' values, we counted the number of configuration options that the value of a variable depends on. A variable can be a compound structure (object or array), whose fields/keys can in turn be other compound structures. Thus, when measuring plugin interactions, we take the nesting levels (or *depths*) of values into account. Specifically, top-level variables are at depth 0; fields/keys of a compound structure at depth k are treated as variables of depth $k+1$. Since a variable's value may change during the execution, we take snapshots of variables' values throughout the execution (every 10,000 executed statements and at the end, totaling 26 snapshots). We record the maximum number of configuration options that each variable's value depends on during the execution.

Figure 14 shows that at all different depths (with a maximum depth of 11), most variables depend on zero or one configuration option. Overall, 88.8 % of variables share the same value in all configurations, and 9.8 % of them have values depending on only one configuration option (column All). We found that high-degree interactions (involving 10-16 plugins) are mostly associated with the variables named wp_filter_id, which are incremented each time a plugin registers a WordPress filter. Since all plugins are optional, its value varies depending on many configuration options.

Sharing inside compound structures (depth > 0) is beneficial if large objects differ only in individual fields. To study the impact of the compact algorithm that enables this inner sharing (Section 3.2.1), we additionally explored the size of the heap without this inner sharing. For 152 out of 158 top-level objects, we would need to

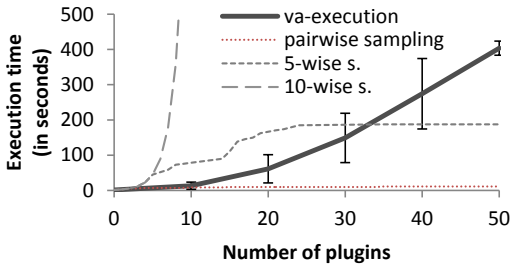

Figure 15: Running time (with standard deviation)

store 774,960 instead of 13,020 fields, whereas we could not even compute the size of 6 top-level objects without inner sharing due to a combinatorial explosion (we waited more than an hour and stopped at about 6,000,000 fields). These results show that the compact algorithm significantly reduces the size of compound structures by exploiting the sharing inside them.

Summary. Overall, the data of our experiment shows that there exist some high-degree plugin interactions, which are hard to catch with random or combinatorial testing. However, we found that the interactions among plugins did not result in severe exponential explosions and that the interactions remain mostly local. Importantly, there exists a significant amount of sharing in terms of computations and data values that motivates variability-execution testing.

5.2 Scalability - RQ2

To evaluate Varex's scalability, we measured the time for running variability-aware execution with different numbers of plugins (wall-clock time). For each number of plugins, we selected 20 random subsets from the set of 50 plugins, except for sets of size 0 and 50 which have only one possible selection. We then ran Varex on these subsets and reported average and standard deviation; for sets of size 0 and 50, we ran it 20 times each. The experiment was carried out on a computer with Intel Core i5 2.30 GHz CPU and 6.00 GB RAM, running Windows 7 64-bit and JVM 7 64-bit with 4.00 GB heap size. We did not even attempt to measure brute-force execution of 2^{50} configurations, but can extrapolate a runtime of 35 million years.

The results are plotted in Figure 15. The standard deviation is high due to the large differences in plugin size and complexity and the random selection of plugins. As seen, performance degrades with additional plugins, due to additional code and interactions, but there is no obvious exponential explosion. The overhead for SAT solving is negligible. Varex executes 20 plugins in about one minute and all 50 plugins (covering 2^{50} configurations) in less than seven minutes. In comparison, executing WordPress with the standard Quercus interpreter took 0.7 second without any plugins and 3.4 seconds with all 50 plugins activated (excluding one conflicting plugin causing a crash, which we discuss next).

In Figure 15, we additionally plot performance for combinatorial testing and brute-force execution (extrapolated based on theoretical bounds [1]). Pairwise testing suffices with small sampling sets that can be executed very quickly. Larger sampling sets are expensive to compute [44], but can still outperform variability-aware execution to some degree. However, in contrast to variability-aware execution, all sampling strategies are necessarily incomplete. Computing n-way samples for large n (such as $n = 16$, as would be needed to guarantee covering all interactions that we detected) approaches brute-force effort, as the sample will have to include at least 2^n configurations.

5.3 Anecdotal Evidence of Plugin Conflicts

Although we did not explicitly search for plugin conflicts (e.g., by writing test cases as outlined in Sections 2 and 4), we found two plugin conflicts that provide anecdotal evidence of the potential of variability-aware execution for testing.

Case 1. *'Undefined function' error caused by plugins 'Contact Form 7 (CON)' and 'Really Simple CAPTCHA (REA)'.* At the end of the variability-aware execution on all 50 plugins, Varex reported an exception occurring in the variability context CON ∧ REA (via rule R7 in Table 1), indicating that a crash occurs at a call to an undefined function win_is_writable in the cleanup function of *REA* when both plugins are activated. Examining the error, we found that when *CON* is also activated, it is the only plugin that calls the cleanup function of *REA*. This interaction causes *REA* to invoke the function win_is_writable, which leads to the crash because WordPress at version 3.4.2 does not yet support it. This error is also confirmed on WordPress website [11].

```
//Plugin REA: really-simple-captcha/really-simple-captcha.php
Line 228: function cleanup (...) {
Line 236: ... win_is_writable ($dir): is_writable ($dir) ...
//Plugin CON: contact-form-7/modules/captcha.php
Line 418: return $wpcf7_captcha->cleanup();
```

Case 2. *Accidental URL overwriting between 'My Calendar (CAL)' and 'WP to Twitter (WPT)'.* After executing all 50 plugins and examining the program's multi-value output, we found that the text displaying an image URL provided by the *CAL* plugin is not a concrete value as expected; instead, it is a Choice between two values depending on configuration option WPT: Choice(WPT, '...plugins/my-calendar/...event.png', '...plugins///my-calendar/...event.png'). That is, the URL does not have its expected value when *WPT* is activated together with *CAL*. This error occurred since the plugins accidentally used the same variable name $wp_plugin_url. When activated, the *WPT* plugin overwrote a different value to the value that was assigned earlier by the *CAL* plugin. This error can be detected by running Varex on a test case as in Figure 4.

```
//Plugin CAL: my-calendar/my-calendar.php
Line 87: $wp_plugin_url = plugin_dir_url ( __FILE__ );
//Plugin WPT: wp-to-twitter/wp-to-twitter.php
Line 39: $wp_plugin_url = plugins_url ();
```

5.4 Threats to Validity

We focused engineering effort to support a single but large-scale and real-world system, because we expect more insights into characteristics of real-world systems than using diverse but smaller or synthetic benchmarks. Although we selected a broadly used system with a typical framework architecture and selected a relatively large set of 'difficult' plugins (popular plugins and plugins that are known to be conflicting), our study was on only the main page of one system with a set of 50 plugins written in PHP; thus, limiting external validity. To cope with the large traces, we had to rely on proxy metrics (measuring statements as computations, sampling snapshots of variable values) which may threaten construct validity. We counted distinct configuration options in formulas as proxy characterizing interactions; most formulas do not contain disjunctions but for those that do we may report slightly higher numbers than interaction-degree metrics used in combinatorial testing [30]. Performance measurements are influenced by JIT compilation and caching effects of the used SAT solver; as a consequence we reported only startup performance by restarting the JVM between every run.

6. RELATED WORK

Variability-aware execution was proposed at least three times independently in the last year. (1) We proposed *variability-aware*

execution in a previous paper [35] and experimented with a hand-written interpreter for the WHILE language and toy examples and experienced the potential for orders-of-magnitude performance improvements over exhaustive brute-force execution. (2) Kim *et al.* independently extended the interpreter of Java Pathfinder into a variability-aware interpreter for Java named *shared execution* [37]. They experimented with small academic product lines (up to 146 configurations) and reported a possible speedup of up to 50 % compared to exhaustive brute-force execution. (3) Austin and Flanagan's proposal of *multiple facets* use a form of variability-aware execution for accurate dynamic information-flow analysis (instead of configuration options they consider different access rights as a reason for tracking alternative values) [13]. They extended a metacircular JavaScript interpreter. On a 300 lines MD5 encryption algorithm with up to 8 configuration options, they demonstrated significant speedups over a sequential brute-force strategy.

Although there are several technical differences among the approaches (e.g., whether to represent variability contexts with propositional formulas or sets, whether and when to merge alternative values), conceptually (with regard to the strategies explained in Section 3) and implementation-wise they are all similar. All prior implementations changed a nonstandard interpreter (which introduced significant interpretative overhead compared to the typical optimized execution environment), and demonstrated speedups only on small examples. The feasibility of these approaches depends on the characteristics of the executed program. An important open question, which we now addressed, is whether such approach scales to a *large real-world scenario*. Our results show that testing plugin-based systems is a promising application that justifies a full-fledged variability-aware execution environment.

In *delta execution* [53], Tucek *et al.* experimented with forking and merging two variants of an instrumented C program, differing in a small patch. However, it is limited to differences between *two* program variants. While they could gain moderate performance improvements, variability-aware execution excels in scenarios with many configurations.

Symbolic Execution. Variability-aware execution is similar to *dynamic symbolic execution* [31, 49, 17] and model checking [22], but has both conceptual and technical differences. The key conceptual difference is that Varex operates on conditional *concrete* values instead of symbolic values. In Varex, a variable may have different values in different configurations, but all values are *concrete*. Configuration options can be viewed as symbolic, but they are used only to map between concrete values and configurations. In contrast to symbolic execution, concrete values never intermix with symbolic ones and we have a clear notion of executing statements conditionally in a variability context. Notice how our PluginConfigOption in Figures 6 and 10 assigns the concrete values True and False depending on a configuration option (v=Choice(ϕ, True, False)).

In symbolic execution and model checking, scalability to large-scale systems remains a challenge [18]. Reisner *et al.* have used dynamic symbolic execution to entirely explore the configuration space of 3 mid-size Java systems [47]. Symbolic execution was expensive in their case, requiring 80 machine weeks for 319 tests in three 10k line applications with less than 30 configuration options each. They do not exploit sharing beyond a common prefix of execution traces.

Variability-Aware Analysis. Variability-aware execution has been inspired by recent work on static analysis of product lines [26, 51, 51, 33, 40, 16, 15, 23, 21]. A community of researchers has investigated how to perform type checking [33, 40, 21, 51], model checking [23, 39], data-flow analysis [16, 15, 40], and other analyses [26, 34] on multiple compile-time configurations of a system at a time. This community has explored how to represent and reason about partial but finite configuration spaces compactly with BDDs or SAT solvers (as used in our variability contexts) [14, 33, 41], how to represent choices of structures [28] and in complex structures [29, 40]. For an overview of the field see a recent survey [52].

Recently, several empirical studies have shown that static analysis of product lines can scale to systems of the size of Linux kernel (over 6000 configuration options in 6 million LOCs) and can outperform some sampling strategies due to the high sharing among the configurations [40, 15, 33]. While these results are encouraging, it was unclear whether they also translate to variability-aware execution, as due to control and data dependencies, we expect more nonlocal effects and interactions among configurations than in type checking or data-flow analysis. Our results confirm this expectation, but also indicate that there is still significant sharing to exploit.

Other Testing Strategies. In product-line testing [45] and framework testing [32] it is a common strategy to unit test components or plug-ins in isolation, while integration tests are often neglected or performed only for specific configurations. Testing product lines is still considered a "rather immature area" [27]. Greiler *et al.* suggest shipping test cases with plug-ins and running them in the configured client system [32]. In essence, this is a strategy that postpones tests of configurations until the configuration is actually used.

Moreover, combinatorial testing allows to compute a set of configurations that cover all combinations among all n-sized sets of configuration options [43, 24]. Pairwise combinatorial testing is efficient to detect all interactions among all pairs of options, but the sample size and effort to compute the sample quickly grows with larger n; sample sets for $n > 5$ are challenging to compute—our example would require $n = 16$ to guarantee full coverage of all actual interactions. Sampling does not need specialized execution environments and can be much faster, as shown in Figure 15, but, by its nature, may miss configuration-related paths. Other methods aim to reduce test cases or configurations via impact analysis [46, 48].

Kim *et al.* and Shi *et al.* have explored static and dynamic analyses to avoid reexecutions of configurations that have exactly the same execution path [36, 38, 50]. They demonstrated in only one large industrial application and mostly small examples for unit tests with few configuration options. In WordPress scenario however, all plugins always influence the execution (even if only because each plugin is initialized); i.e., such analysis could not exclude any configuration, resulting in a brute-force approach.

In several scenarios multiple program variants are executed in parallel for security reasons [25, 20]. Executions are synchronized, but similarities among variants are not exploited.

7. CONCLUSIONS

Variability-aware execution has been proposed recently to improve performance over brute-force execution for testing configurable systems; however, it has been demonstrated on only small examples. In this paper, we addressed the question of whether such an approach can scale to large real-world scenarios. Running our variability-aware PHP interpreter Varex on the WordPress web application with 2^{50} configurations, we found that there exists a significant amount of sharing across plugin configurations, which allows Varex to scale. The results showed that developing variability-aware execution environments for testing is a promising direction.

8. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation (NSF) CCF-1018600, CNS-1223828, CCF-1318808, CCF-1349153, and CCF-1320578 grants.

9. REFERENCES

[1] Covering array tables. http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.

[2] Eclipse website. http://www.eclipse.org/.

[3] JavaBDD website. http://javabdd.sourceforge.net/.

[4] Microsoft Office website. http://office.microsoft.com/.

[5] Mozzila add-ons website. https://developer.mozilla.org/en-US/addons.

[6] Plugin conflicts. http://wiki.simple-press.com/installation/troubleshooting/plugin-conflicts/.

[7] Quercus website. http://quercus.caucho.com/.

[8] Varex website. http://home.engineering.iastate.edu/~hungnv/Research/Varex/.

[9] Web site test tools and site management tools. http://www.softwareqatest.com/qatweb1.html.

[10] WordPress plugin compatibility beta. http://wordpress.org/news/2009/10/plugin-compatibility-beta/.

[11] WordPress support website. http://wordpress.org/support/topic/call-to-undefined-function-win_is_writable-on-line-236.

[12] WordPress website. http://wordpress.org/.

[13] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 165–178, New York, 2012. ACM Press.

[14] D. Batory. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20, Berlin/Heidelberg, 2005. Springer-Verlag.

[15] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364, New York, 2013. ACM Press.

[16] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24, New York, 2012. ACM Press.

[17] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 209–224, Berkeley, CA, 2008. USENIX Association.

[18] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1066–1071, New York, 2011. ACM Press.

[19] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[20] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 322–331, Washington, DC, 2008. IEEE.

[21] S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2013.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[23] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, 2010. ACM Press.

[24] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 129–139, New York, 2007. ACM Press.

[25] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. USENIX Security Symposium (USENIX-SS)*, Berkeley, CA, USA, 2006. USENIX Association.

[26] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, 2006. ACM.

[27] E. Engström and P. Runeson. Software product line testing - A systematic mapping study. *Information and Software Technology (IST)*, 53(1):2–13, 2011.

[28] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, 2011.

[29] M. Erwig and E. Walkingshaw. Variation programming with the choice calculus. In *Generative and Transformational Techniques in Software Engineering IV*, pages 55–100. Springer Berlin Heidelberg, 2013.

[30] B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 90–99, Los Alamitos, CA, 2011. IEEE Computer Society.

[31] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 213–223, New York, 2005. ACM Press.

[32] M. Greiler, A. van Deursen, and M.-A. Storey. Test confessions: A study of testing practices for plug-in systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 244–254, Piscataway, NJ, 2012. IEEE Press.

[33] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):14:1–14:39, 2012.

[34] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, 2011. ACM Press.

[35] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8, New York, 2012. ACM Press.

[36] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57–68, New York, 2011. ACM Press.

[37] C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 221–230, Los Alamitos, CA, 2012. IEEE Computer Society.

[38] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 257–267, New York, 2013. ACM Press.

[39] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280, Los Alamitos, CA, 2009. IEEE Computer Society.

[40] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, New York, 2013. ACM Press.

[41] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240, New York, 2009. ACM Press.

[42] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction: The security threat from within software systems. *Progress in Informatics*, pages 75–89, 2008.

[43] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, 2011.

[44] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 26–36, New York, 2013. ACM Press.

[45] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg, 2005.

[46] X. Qu, M. Acharya, and B. Robinson. Impact analysis of configuration changes for test case selection. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 140–149, Washington, DC, 2011. IEEE Computer Society.

[47] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 445–454, New York, 2010. ACM Press.

[48] B. Robinson, M. Acharya, and X. Qu. Configuration selection using code change impact analysis for regression testing. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 129–138, Washington, DC, 2012. IEEE Computer Society.

[49] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, New York, 2005. ACM Press.

[50] J. Shi, M. Cohen, and M. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 270–284, Berlin/Heidelberg, 2012. Springer-Verlag.

[51] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, 2007. ACM Press.

[52] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014. to appear.

[53] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 193–204, New York, 2009. ACM Press.