Legato: An At-Most-Once Analysis with Applications to Dynamic Configuration Updates

John Toman
Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
jtoman@cs.washington.edu

Dan Grossman
Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
djg@cs.washington.edu

Abstract

Modern software increasingly relies on external resources whose location or content can change during program execution. Examples of such resources include remote network hosts, database entries, dynamically updated configuration options, etc. Long running, adaptable programs must handle these changes gracefully and correctly. Dealing with all possible resource update scenarios is difficult to get right, especially if, as is common, external resources can be modified without prior warning by code and/or users outside of the application's direct control. If a resource unexpectedly changes during a computation, an application may observe multiple, inconsistent states of the resource, leading to incorrect program behavior.

This paper presents a sound and precise static analysis, Legato, that verifies programs correctly handle changes in external resources. Our analysis ensures that every value computed by an application reflects a single, consistent version of every external resource's state. Although consistent computation in the presence of concurrent resource updates is fundamentally a concurrency issue, our analysis relies on the novel *at-most-once* condition to avoid explicitly reasoning about concurrency. The at-most-once condition requires that all values depend on at most one access of each resource. Our analysis is flow-, field-, and context-sensitive. It scales to real-world Java programs while producing a moderate number of false positives. We applied Legato to 10 applications with dynamically updated configurations, and found several non-trivial consistency bugs.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, Dynamic Configuration Updates

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.24

Supplement Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.4.3.2`

Funding This paper is based upon work sponsored in part by DARPA under agreement number FA8750-16-2-0032.

Acknowledgements The authors would like to thank Doug Woos, Chandrakana Nandi, Emina Torlak, Manuel Fahndrich, Francesco Logozzo, and Christian Kästner for their comments on early drafts of this work. We would also like to thank the anonymous reviewers for their feedback.

1 Introduction

Programs are no longer monolithic collections of code. In addition to source code, modern applications consist of configuration files, databases, network resources, and more. Treating these *external resources* as static inputs to the program is infeasible for adaptable, long

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 24; pp. 24:1–24:32
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
if(hasReadPermission("harmless_file")) {
   open("harmless_file").read();
}
```

Figure 1 Example time-of-check-to-time-of-use bug caused by a dynamic resource update: if `"harmless_file"` is replaced with a symlink to another user's file after the permissions check but before the `open()` call, a leak of another user's private information will occur.

running programs. Remote hosts may become unavailable or change their APIs, database entries may be changed by other threads or programs, the filesystem may be changed by other tenants on the program's host, and users may update the configuration options of the program. We refer to these changing, evolving resources as *dynamic* external resources; together, these dynamic external resources form an application's view of the dynamic environment in which it executes.

Correctly handling changes in these external resources is challenging. If a dynamic resource is changed between two accesses used in the same computation, a program can observe two or more inconsistent versions of the resource state, which can lead to arbitrary and often incorrect behavior. For example, Figure 1 contains a program fragment exhibiting a well-known time-of-check-to-time-of-use defect [36, 6, 31] that can lead to a malicious user circumventing filesystem permissions. The attack is possible precisely because the code in question can observe two versions of the filesystem state: specifically, two different versions of the read permission. Such issues are not restricted to filesystems; similar problems can be found in applications that interact with databases with multiple users [1] or that support instantaneous configuration updates [43].

Further complicating matters, unlike state under sole control of the program, external resources are often mutated by other programs or users of the system without warning and it is often impossible for the application to prevent such changes. Errors due to dynamic resource updates are difficult to anticipate ahead of time, and (like concurrency errors) require difficult-to-write functional tests to manually uncover. Further, although the example shown in Figure 1 can be detected with a simple syntactic analysis, the dynamic resource errors we have found in practice often involve multiple levels of indirection through the heap and flows through multiple method calls. There has been extensive work to help programmers contend with and correctly handle these changes [4, 31, 9, 5]. However, existing techniques take a piecemeal approach tailored to a specific resource type (e.g., files [36, 6], configuration options [43], etc.).

This paper presents a unified approach to verify that programs always observe consistent versions of external resource state. Key to our approach is the *at-most-once* condition. The at-most-once condition states that a value may depend on at most one access of each external resource. Intuitively, programs observe inconsistent resource states when a resource changes between two or more related accesses of a resource. By restricting all computations to at most one access per resource, the condition guarantees that every value computed by the program always reflects some consistent snapshot of each resource's state.

We efficiently check this condition for complex, real-world programs using a novel static analysis. Conceptually, our analysis versions the external resources accessed by a program so that each read of a resource is assigned a unique version. Our analysis tracks these versioned values as they flow through the program and reports when two or more distinct versions flow to the same value. Although our analysis focuses on errors caused by concurrent changes it does *not* explicitly reason about concurrency involving external updates. Our

```
int getDoubled() {
  return Config.get("number") +
    Config.get("number");
}
```

Figure 2 Example of inconsistency due to dynamically updated configuration options. If the `"number"` configuration option changes between the two calls to `Config.get()`, a non-even number may be returned.

```
int a = Config.get("number");
int b = 0;
while(⋆) {
  b += a;
}
```

Figure 3 Example of a resource used multiple times after being read. $\star$ represents a side-effect free, uninterpreted loop condition. This use pattern is correct because the `"number"` resource is accessed only once in computing `b`.

analysis is interprocedural and scales to large programs. The analysis is flow-, field-, and context-sensitive, and can accurately model dynamic dispatch.

We implemented the Legato[1] analysis as a prototype tool for Java programs. We evaluated Legato on 10 real-world Java applications that use dynamic resources. These applications were non-trivial: one application in our evaluation contains over 10,000 methods. Legato found 65 bugs, some of which caused serious errors in these applications. Further, we found that the at-most-once condition is a good fit for real applications that use external resources: violations of the at-most-once condition reported by our analysis often corresponded to bugs in the program. Legato had a manageable ratio of true and false positives. Our tool is also efficient: it has moderate memory requirements, and completed in less than one minute for 6 out the of 10 applications in our benchmark suite.

In summary, this paper makes the following contributions:

- We define the at-most-once condition, a novel condition to ensure consistent usage of external resources (Section 2).
- We present a novel static analysis for efficiently checking the at-most-once condition (Sections 3 and 4).
- We describe Legato, an implementation of this analysis for Java programs (Section 5).
- We show that Legato can find real bugs in applications that use dynamic resources (Section 6).

2 At-Most-Once Problems

Legato targets programs that use *dynamic external resources*. Unlike static program resources (e.g., program code, constant pools, etc.) dynamic resources are statically identifiable entities that may be changed without warning by code or programs outside of an application's control. In the presence of external changes, programs may observe inconsistent versions of an external resource's state.

For example, Figure 2 shows a (contrived) example of an error due to dynamically updated configuration options. Although callers of `getDoubled()` would reasonably expect the function to always produce an even number, an update of the `"number"` option between the two calls to `Config.get()` may result in an odd number being returned. This unexpected behavior occurs because the application observes inconsistent versions of `"number"`. The time-of-check-to-time-of-use error in Figure 1 from the introduction is another example.

One possible technique for detecting these errors is to concretely model dynamic resource updates and reason explicitly about update/access interleavings. Unfortunately, explicitly

[1] Legato is open-source, available at `https://github.com/uwplse/legato`

modeling concurrency, e.g., [11, 3, 10], is intractably expensive on large programs or requires specific access patterns [25, 27].

Legato instead verifies consistent usage of dynamic resources without explicitly reasoning about concurrent updates and reads. In the worst case, a resource may change between *every* access; i.e., every access may yield a unique version of the resource. For example, suppose that the configuration accessed in Figure 2 is updated by another thread in response to user input. In the presence of non-deterministic thread scheduling and without prior synchronization between the two accesses of `"number"` on lines 2 and 3, the option may be updated some arbitrary number of times. The current implementation of `getDoubled` correctly handles updates that occur before or after the two accesses: only interleaved updates are problematic.

A key insight of Legato is that a program that is correct under the worst-case resource update pattern described above will necessarily be correct under *any* update pattern. Further, under the assumption that every access yields a distinct version of the underlying resource, values from two or more different accesses of the same resource can never be combined without potentially yielding an inconsistent result. It is therefore sufficient to verify that a value depends on *at most one* access to each resource. Verifying this condition for all values in a program is the *at-most-once problem.*

The at-most-once problem places no restrictions on the number of times a resource may be *used* once read, nor how many times a resource may be accessed, only on how many times the resource may be accessed in computing a single value. For example, the code in Figure 3 is correct according to our definition of at-most-once. When the `"number"` option is read on line 1 it reflect a single, consistent version of the option at the time of read. Although `"number"` may be updated an arbitrary number of times as the loop executes, the value of `a` is unaffected by these updates and remains consistent as it is used multiple times during the execution of the loop. As a result, after the loop finishes, the value of `b` will reflect a consistent version of the `"number"` option. If the body of the loop was `b += Config.get("number")`, the at-most-once requirement would be violated.

3 The Legato Analysis

Legato is a whole-program dataflow analysis for detecting at-most-once violations in programs that use dynamic resources. For ease of presentation, throughout the rest of this paper, we assume that there is only one resource of interest that is accessed with the function `get()`. The analysis described here naturally extends pointwise to handle multi-resource scenarios. Conceptually, the analysis operates by assigning a globally unique, abstract version to the values returned from each resource access. If two or more unique versions flow to the same value, this indicates that a resource was accessed multiple times, thus violating at-most-once.

In a dynamic setting, every read of a resource can be tagged with an automatically incrementing version number. With this approach, detecting violations of at-most-once is straightforward: when two or more different version numbers reach the same value, at-most-once *must* have been violated. This is the approach taken by Staccato [43], which finds inconsistent usage of dynamic configuration options. However, concrete version numbers do not translate to the static setting.

In place of concrete numbers, resource versions can be *abstractly* represented by the site at which a resource was accessed and the point in the program execution that the resource occurred. The presence of uninterpreted branch and loop conditions makes it impossible to determine the absolute point in a program execution at which a resource access occurs. Instead, Legato uses *abstract resource versions* (ARVs) to encode accesses *relative* to the current point in the program execution. For example, an ARV can represent "the value returned from the 2nd most recent execution of the statement s", which precisely identifies a single access while remaining agnostic about the *absolute* point in the program execution the access occurred.

The Legato analysis combines a reachability analysis with the abstract domain of ARVs to discover which resource versions flow to a value. The ARV lattice is designed such that the meet of two ARV representing different accesses (and therefore versions) yields $\bot$, which indicates a possible violation of at-most-once.

We first present a simple intraprocedural analysis that does not support loops, heap accesses, or method calls (Section 3.2). We then extend the approach to handle loops (Section 3.3). The transformers defined by these two sections illustrate the core Legato analysis. In principle, this basic analysis could be extended to extremely conservatively handle language features, such as the heap or methods. However, in practice, doing so would result in enormous precision loss. We therefore show how we extend the analysis to field- and flow-sensitively handle information flow through the heap (Section 3.5). Extending the analysis to precisely handle method calls is non-trivial, and is discussed in Section 4. Other program features (e.g., exceptions, arrays, etc.) are straightforward extensions of the ideas presented here.

Abstract Resource Versions. As mentioned above, an abstract resource version (ARV) represents a resource version by the access site and the point in time at which the access was performed. To ensure soundness, values returned from different resource accesses must be assigned unique ARVs (we expand on this point further in Section 3.4). In a simple language with no loops or methods, ARVs are simply expression labels: each label represents the unique value produced by the execution of the labeled expression. In the presence of loops, we augment these labels with a priming mechanism to differentiate between multiple executions of the same expression. To precisely handle methods, in Section 4.1 we generalize to strings of primed labels, which identify an access by the sequence of method calls taken to reach an access site (similar to the approach taken by [48]). Finally, in Section 4.2, we further generalize ARVs to sets of strings (represented as tries) to encode multiple possible accesses that may reach a program value. However, even with this representation, our analysis always maintains the invariant that each ARV abstracts a single, unique resource access.

3.1 Preliminaries

Before describing the analysis, we first briefly review some relevant background information.

IDE. The Legato analysis uses the IDE (Interprocedural Distributive Environment) program analysis framework [40]. The IDE framework can efficiently solve program analysis problems stated in terms of *environment transformers*. An environment is a mapping from dataflow symbols (e.g., variables) to values. The domain of symbols must be finite, but the domain of values may be infinite, provided the values form a finite-height, complete lattice. The meet of environments is performed pointwise by symbol. IDE analyses assign environment transformers to edges in the program control-flow graph. However, to aid exposition, throughout the remainder of this section we will instead denote *statements* into

$$
\begin{array}{llll}
\text{(const)} & c & ::= & 0 \mid 1 \mid \ldots \\
\text{(var)} & v & ::= & \texttt{a} \mid \texttt{b} \mid \ldots \\
\text{(atom)} & a & ::= & c \mid v \\
\text{(expr)} & e & ::= & a + a \mid a \mid \texttt{get}^{\ell}\texttt{()} \\
\text{(stmt)} & s & ::= & v = e \mid s; s \mid \texttt{skip} \\
 & & & \mid \ \texttt{if}\ a\ \texttt{then}\ s_1\ \texttt{else}\ s_2
\end{array}
$$

Figure 4 Grammar for the loop-, and method-free language.

Statement		**Transformer**
$[\![v_1 = v_2]\!]$	$\triangleq$	$\lambda e.e[v_1 \mapsto e(v_2)]$
$[\![v = c]\!]$	$\triangleq$	$\lambda e.e[v \mapsto \top]$
$[\![v_1 = v_2 + v_3]\!]$	$\triangleq$	$\lambda e.e[v_1 \mapsto e(v_2) \sqcap e(v_3)]$
$[\![v = \texttt{get}^{\ell}\texttt{()}]\!]$	$\triangleq$	$\lambda e.e[v \mapsto \widehat{\ell}]$
$[\![\ \texttt{skip}\ ]\!]$	$\triangleq$	$\lambda e.e$
$[\![\ \texttt{if}\ a\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]$	$\triangleq$	$\lambda e.([\![s_1]\!]e) \sqcap ([\![s_2]\!]e)$
$[\![s_1; s_2]\!]$	$\triangleq$	$\lambda e.[\![s_2]\!]([\![s_1]\!]e)$

Figure 5 Environment transformers of the basic analysis.

environment transformers.[2]

The IDE framework targets a specific subclass of analyses where the environment transformers distribute over the meet operator on environments. That is, for all transformers $t : \mathit{Env} \to \mathit{Env}$ and all environments $e_1, e_2, \ldots, e_n$, $\bigsqcap_i t(e_i) = t(\bigsqcap_i e_i)$, where equality on environments is defined pointwise. Given a set of distributive environment transformers, the IDE framework produces a flow and context-sensitive analysis with polynomial time complexity.

Access Paths. An *access path* [15, 19] is an abstract description of a heap location. An access path consists of a local variable v, and a (potentially empty) sequence of field names $f.g.h\ldots$ Together, these two elements name the location reachable from v through fields $f, g, h, \ldots$ We will write ϵ to represent and empty sequence of fields, π to refer to an arbitrary (potentially empty) sequence of fields, and $v.\pi$ to denote an arbitrary access path.

3.2 The Basic Analysis

We first present our analysis on a limited language described by the grammar in Figure 4. Every call to `get()` is uniquely labeled with ℓ: we will write concrete labels as 1, 2, etc. Our basic language contains no looping constructs, as a result every `get()` expression is executed at most once. Thus, every access can be uniquely identified by the label of a `get()` expression. For this language, the abstract resource versions are `get()` expression labels: $\widehat{\ell}$ represents the

[2] This change in presentation does not change the behavior of the analysis; the denotation of a statement is a simplification of the meet of the composition of the edge transformers for all paths through a statement.

unique version of the resource returned by the corresponding $\texttt{get}^{\ell}\texttt{()}$ expression. Further, at this point the analysis operates on access-paths with no field sequences: we abbreviate $v.\epsilon$ as v.

The basic LEGATO analysis is expressed using the environment transformers in Figure 5. Conceptually, for a program s, the analysis applies the empty environment (i.e., all facts map to $\top$) to the transformer associated with statement s. Thus, the analysis result is given by $[\![s]\!]\ (\lambda\_.\top)$. The analysis is standard in its handling of several language features. For instance, sequential composition of statements is modeled by composing environment transformers, and conditional statements are modeled by taking the meet of the environments yielded from both branches.

The interesting portion of the analysis lies in the handling of variable assignments. Assignments overwrite previous mappings in the environment of the left-hand side with the abstract value of the right-hand side. Integer constants are never derived from resources, and therefore have the abstract value $\top$, which represents any value not derived from a resource. The statement $v_1 = v_2$ associates v_1 with the abstract version contained in v_2. Resource accesses have the abstract value $\widehat{\ell}$ which, as discussed above, is sufficient to uniquely identify the value returned from the access $\texttt{get}^{\ell}\texttt{()}$. This simplified version of the LEGATO analysis is very similar in style to a constant propagation analysis, where in place of integers or booleans, the constants of interest are abstract resource versions.

Values may become inconsistent for two reasons. The first is due to the addition operator. The expression $v_1 + v_2$ is given the abstract value $e(v_1) \sqcap e(v_2)$. The meet operator for these ARVs is derived from a flat lattice:

$$\top \sqcap x = x \qquad x \sqcap \bot = \bot \qquad \widehat{i} \sqcap \widehat{i} = \widehat{i} \qquad \widehat{i} \sqcap \widehat{j} = \bot, \text{ if } i \neq j$$

where i and j are two arbitrary labels. If $e(v_1) = \widehat{i}$ and $e(v_2) = \widehat{i}$, then the result of the addition still depends only on the resource accessed at $\texttt{get}^{i}\texttt{()}$. In this case, at-most-once is not violated, and the meet yields $\widehat{i}$ as the abstract value of the overall expression. However, if $e(v_1) = \widehat{i}$ and $e(v_2) = \widehat{j}$ then the program is combining two unique versions of the resource, which violates at-most-once. The meet of these two incompatible versions yields $\bot$, which is the "inconsistent" value in the lattice.

Finally, a variable may be assigned $\bot$ due to LEGATO's conservative handling of conditional statements. Recall that the environments produced by the two branches of a `if` statement are met at the control-flow join point of the conditional. Thus, if a variable x is mapped to two distinct, non-$\bot$ values in environments produced by different branches of a conditional, those values will be met yielding $\bot$. In this case, the result of $\bot$ does not correspond to a violation of at-most-once, and is a false positive. The alternative, full path-sensitivity, is unacceptably expensive. We do support a limited form of path-sensitivity to precisely model dynamic dispatch (Section 4.2).

3.3 Loops

The simple analysis presented so far is no longer sound if we extend the language with loop statements:

$$\mathit{stmt} \quad ::= \quad \ldots \mid \texttt{while}\ a\ \texttt{do}\ s\ \texttt{end}$$

If a $\texttt{get}^{\ell}\texttt{()}$ expression is in a loop, each evaluation of $\texttt{get}^{\ell}\texttt{()}$ must be treated as returning a unique version. However, the transformers presented in the previous section effectively assume $\texttt{get}^{\ell}\texttt{()}$ always returns the same version. We therefore extend the transformers and lattice to distinguish resource accesses from distinct iterations of an enclosing loop.

```
a = 1; // a: ⊤
b = get¹(); // b: 1̂
c = get²(); // c: 2̂
d = a + b; // d: 1̂
e = c + d; // e: ⊥
```

Figure 6 Results of the basic analysis. The comments on each line show the abstract value assigned to the variable assigned on that line.

```
1 while ⋆ do
2   b = a; // a: 1̂, b: 1̂
3   a = get¹() // a: 1̂, b: 1̂′
4 end
5 c = a + b; // c: ⊥
```

Figure 7 Example of priming due to loops. The abstract values shown in comments are derived after executing the loop once. On line 3 LEGATO primes the abstract value of `b` to distinguish it from the fresh value returned by `get`^{1}`()`.

In a dynamic setting, we could associate every resource access `get`$^\ell$`()` with a concrete counter c_ℓ incremented on every execution of `get`$^\ell$`()`. In this (hypothetical) scenario, `get`$^\ell$`()` yields the abstract version, $\langle \ell, c_\ell \rangle$: by auto-incrementing c_ℓ the analysis ensures executions of `get`$^\ell$`()` from different iterations are given unique abstract versions.

This straightforward approach fails in the static setting: without *a priori* knowledge about how many times each loop executes, the analysis would fail to terminate. We introduce *priming* to address this issue. A primed `get()` label $\widehat{\ell}^n$ represents the $n{+}1^{th}$ most recent access of the resource at `get`$^\ell$`()`. For example, $\widehat{1}''$ (i.e., $\widehat{1}^2$) represents the unique value produced by the third most recent evaluation of `get`^{1}`()`, and $\widehat{2}$ (i.e., $\widehat{2}^0$) is the value returned from the most recent evaluation of `get`^{2}`()`. Abstract versions with the same base label but differing primes are considered unique from one another in the lattice, i.e., $\widehat{i}^n \sqcap \widehat{j}^m = \bot \iff i \neq j \vee m \neq n$. Thus, this domain distinguishes between accesses due to different `get()` expressions as well as different invocations of the *same* `get()` expression.

The syntactic structure of loops are handled using a standard fixpoint technique. The addition of loops changes how $v =$ `get`$^\ell$`()` statements are handled in the analysis. As before, the variable v is assigned the abstract value $\widehat{\ell}$. In addition, a prime is added to all *existing* abstract values with the base label ℓ. We extend the environment transformer for the $v =$ `get`$^\ell$`()` case in Figure 5 as follows:

$$\lambda e.\lambda v'. \begin{cases} \widehat{\ell} & \text{if } v \equiv v' \qquad (1) \\ \widehat{\ell}^{n+1} & \text{if } e(v') \equiv \widehat{\ell}^n \qquad (2) \\ e(v') & o.w. \qquad (3) \end{cases}$$

In other words, the $v =$ `get`$^\ell$`()` statement creates a new environment[3] such that:

1. v maps to $\widehat{\ell}$, i.e., the most recent version returned from `get`$^\ell$`()`
2. Variables besides v that map to the base label ℓ have a prime added, indicating these values originate one more invocation of `get`$^\ell$`()` in the past
3. All other variables retain their value from *env*

A program illustrating this behavior is shown in Figure 7.

Termination. It is not obvious that the above environment transformer will not add primes forever. We therefore informally argue for termination.

Let us consider the simple case with a single loop and one call to `get()` labeled ℓ. Variables that are definitely not assigned a value from `get`$^\ell$`()` will not be primed and therefore do not

[3] Recall that an environment is a mapping of symbols (in this case, variables) to abstract values: the function term $\lambda v'. \cdots$ is such an environment.

affect achieving fixpoint. For variables to which $\texttt{get}^\ell\texttt{()}$ *may* flow, the flow occurs along some single chain of assignments, e.g. $\texttt{a = get}^\ell\texttt{(); b = a; c = b; ...}$ If instead the assignment occurred along multiple possible chains, the conservative handling of conditionals will yield $\bot$, ensuring the analysis achieves fixpoint.

Consider now the case where some assignments in the chain occur *conditionally*, e.g.:

```
1 while ⋆ do
2   if ⋆ then b = a else skip;
3   a = get^1()
4 end
```

where $\star$ represents uninterpreted loop and branch conditions. In this example, `b` receives the value of some arbitrary previous invocation of $\texttt{get}^1\texttt{()}$. Our domain of primed labels cannot precisely represent this value, but the analysis will conservatively derive $\bot$ for `b`, again ensuring the analysis achieves fixpoint. After two iterations of the analysis, two possible values for `b`, $\widehat{1}$ and $\widehat{1}'$, will flow to line 3 from the two branches of the conditional on the previous line. The meet of these two values is $\bot$.

The last case to consider in the single loop case is a chain of definite assignments from $\texttt{get}^\ell\texttt{()}$ to some variable v. For some chain of length k, it is easy to show that the resource will propagate along the chain in at most k analysis iterations. Thus, the resource will flow over the `get()` expression at most k times, and receive at most k primes. After fully propagating along the chain, the value in v will not receive further primes: on further iterations of the analysis the value in v is killed by the previous definite assignment in the chain.

Finally, we consider nested loops. As a representative case, consider the following scenario:

```
1 while ⋆ do
2   b = a;
3   while ⋆ do  a = get^1() end
4 end
```

Other possible combinations of assignments and nesting generalize straightforwardly from this example. After the first pass through the outer-loop, the environment produced is $[\texttt{a} \mapsto \widehat{1}]$. On the second pass, the environment that reaches line 3 is $[\texttt{a} \mapsto \widehat{1}, \texttt{b} \mapsto \widehat{1}]$. One further iteration of the inner-loop produces $[\texttt{a} \mapsto \widehat{1}, \texttt{b} \mapsto \widehat{1}']$. The meet of this environment with the previous input environment on line 3 assigns `b` the value $\bot$, ensuring a fixpoint is reached.

An alternative approach would be to artificially limit the number of primes on a label to some small constant k. However, we decided against choosing an *a priori* bound for the number of primes lest this bound introduce false positives. However, we found in practice we needed at most 2 primes for the programs in our evaluation set. This finding is consistent with Naik's experience with abstract loop vectors [35], which are similar to our priming approach.

3.4 Soundness

We have proved that the core analysis presented is sound. We first defined an instrumented concrete semantics that: 1) assigns to each value returned from `get()` a unique, concrete version number, and 2) for each value, collects the set of concrete resource versions used to construct that value. The concrete semantics only considers direct data dependencies when collecting the versions used to construct a given value. We define soundness in relation to these concrete semantics. The LEGATO analysis is sound if, whenever variable is derived from multiple concrete versions in any execution of the instrumented semantics, the analysis derives $\bot$ for that variable. As our concrete semantics uses only direct dependencies for

Statement		Transformer
$\llbracket v.f = c \mid \texttt{null} \rrbracket$	$\triangleq$	$\lambda env.env[\mathit{pref}(v.f) \mapsto \top]$
$\llbracket v = \texttt{null} \mid \texttt{new}\ \ \texttt{T} \mid c \rrbracket$	$\triangleq$	$\lambda env.env[\mathit{pref}(v) \mapsto \top]$
$\llbracket v_1 = v_2.f \rrbracket$	$\triangleq$	$\lambda env.env[\mathit{pref}(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.f.\pi)]$
$\llbracket v_1 = v_2 \rrbracket$	$\triangleq$	$\lambda env[\mathit{pref}(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.\pi)]$

Figure 8 New environment transformers for the heap. The $\mathit{pref}(x)$ function yield the set of all access paths in e with x as a prefix. In addition, all references π are implicitly universally quantified.

collecting version numbers, our soundness claim is only with respect to such dependencies and ignores information propagated via control-flow. We discuss this reasons for this choice further in Section 5.4.

Our proof of soundness relies on a *distinctness invariant*: two variables have different abstract resource versions if they have different concrete version numbers under the concrete semantics. In other words, when two variables have the same abstract version, they *must* be derived from the same resource access in all possible program executions. Thus, the invariant ensures that when values derived from different concrete resource versions are combined by a program, the analysis will take the meet of distinct abstract resource versions yielding $\bot$. The converse is also true: if two values with the same abstract resource version are combined, then no program execution will combine two values derived from distinct resource accesses.

The justifications given above for the environment transformers and analysis domain provide intuitive arguments for why this invariant is maintained. The full proofs and concrete semantics are omitted for space reasons: they are included in the appendix of the paper. Although our proof is stated only for the simple intraprocedural analysis presented so far, when we extend the analysis to support methods in Section 4 we provide an argument for the preservation of the distinctness invariant.

3.5 Fields and the Heap

We now consider a language with objects and fields.

$$\begin{array}{rcl} \mathit{expr} & ::= & \ldots \mid \texttt{new}\ \ \texttt{T} \mid v.f \\ \mathit{atom} & ::= & \ldots \mid \texttt{null} \\ \mathit{stmt} & ::= & \ldots \mid v.f = a \end{array}$$

A subset of the new environment transformers for the heap language are given in Figure 8. In this version of the language, our transformers operate on access paths with non-empty field sequences as opposed to plain variables. These environment transformers encode the effect of each statement on the heap: for example, constants, `null`, and `new` expressions on the right hand side of an assignment "kill" access-paths reachable from the left-hand side.

There are two statement forms that require special care that do not appear in Figure 8. First, Legato handles assignments with a `get()` right-hand side with the environment transformer from Section 3.3 extended to support access paths instead of variables. For an assignment of the form $v = \texttt{get}^{\ell}\texttt{()}$, Legato uses the following transformer:[4]

$$\lambda e.\lambda\langle v'.\pi\rangle.\begin{cases}\widehat{\ell} & \text{if } v' \equiv v\\ \widehat{\ell}^{n+1} & \text{if } e(v'.\pi) \equiv \widehat{\ell}^{n} \wedge v' \not\equiv v\\ e(v'.\pi) & o.w.\end{cases}$$

The second statement form, heap *writes* such as $v_1.f = v_2$, is handled conservatively. Legato uses strong updates only for access paths with the $v_1.f$ prefix. After the heap-write, the abstract value reachable from some access path $v_1.f.\pi$ is precisely the value reachable from $v_2.\pi$. However, an access path that only *may* alias with $v_1.f$ is weakly updated. A weak update of the access path $v_3.\pi'$ to the abstract value $\widehat{\ell}^{n}$ takes the meet of the current value of $v_3.\pi'$ with $\widehat{\ell}^{n}$. Given the definition of the label lattice, this treatment of weak updates means that an access-path $v.\pi$ "updated" via aliasing cannot be updated at all: the new value must exactly match the existing value of the access-path or the access-path may have no value at all, represented by $\top$.

Formally, Legato assigns a heap write statement $v_1.f = v_2$ the environment transformer:

$$\lambda e.\lambda\langle v.\pi\rangle.\begin{cases}e(v_2.\pi') & \text{if } v.\pi \equiv v_1.f.\pi'\\ e(v_2.\pi') \sqcap e(v.\pi) & \text{if } v.\pi \not\equiv v_1.f.\pi' \wedge \mathit{mayAlias}(v.\pi, v_1.f.\pi')\\ e(v.\pi) & o.w.\end{cases}$$

Resolving the *mayAlias* query is an orthogonal concern to the Legato analysis. In our implementation we use an off-the-shelf, interprocedural, flow- and context-sensitive may alias analysis (Section 5.1).

4 Interprocedural Analysis

The interprocedural version of Legato is a non-trivial extension of the intraprocedural analysis from the previous section. There are two main extensions to the core analysis. First, Legato soundly accounts for *transitive* resource accesses. A transitive resource access refers to when a method `m()` returns the result of an invocation of $\texttt{get}^{\ell}\texttt{()}$; the analysis must distinguish between abstract values produced by separate invocations of `m()`. In addition, to analyze realistic Java code, Legato precisely models dynamic dispatch. If a method call `m()` may dynamically dispatch to one of several possible implementations, Legato soundly combines the unique abstract values returned by each implementation without sacrificing precision.

Definitions. For presentation purposes only, we begin by making the simplifying assumption that all methods are static (i.e., all call sites have one unique callee), a method has a single formal parameter p, all methods end with a single `return` statement, and method calls are always on the right hand side of an assignment. We extend the grammar for expressions and statements as follows:

$$\mathit{expr} ::= \ldots \mid m^{k}(a) \qquad\qquad \mathit{stmt} ::= \ldots \mid \texttt{return}\ a$$

[4] In our formalism, we assume that `get()` returns a primitive value, and thus the environment will only contain mappings for $v.\epsilon$.

Transformers for a method invocation: $v_1 = m^k(v_2) \to m(p)\{\ldots; \texttt{return}\ r\}$		
Call-to-start	**Exit-to-return**	**Call-to-return**
$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} e(v_2.\pi) & \text{if } v' \equiv p \\ \top & o.w. \end{cases}$	$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} \rho(e(p.\pi)) & \text{if } v' \equiv v_2 \\ & \wedge \pi \not\equiv \epsilon \\ \rho(e(r.\pi)) & \text{if } v' \equiv v_1 \\ \top & o.w. \end{cases}$	$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} \top & \text{if } v' \equiv v_1 \\ \top & \text{if } v' \equiv v_2 \\ & \wedge \pi \not\equiv \epsilon \\ \tau(e(v'.\pi)) & o.w. \end{cases}$

Figure 9 Interprocedural environment transformers. The names of the columns correspond to the transformer names in the original IDE paper. ρ is a function that transforms values that flow out of a method. τ transforms values propagated over method calls. We will define these methods later in the section. The analysis allows for strong updates to heap locations reachable from the argument of a method, although the base variable retains its value from the caller.

```
1 m() {
2   while ⋆ do a = get^1() end;
3   return a
4 }
5 b = m^2();
6 c = m^3()
```

Figure 10 A non-trivial interprocedural resource access.

All method calls are labeled: these sets of labels do not overlap with `get()` expression labels. We will continue to use ℓ to denote an arbitrary `get()` expression label, and k to denote a call site label. We will use the same notation for method call labels used in ARVs (i.e., $\widehat{1}$) as we did for `get()` labels in Section 3: context will make clear which type of label we mean.

The interprocedural environment transformers used by Legato are mostly standard in the mapping of dataflow symbols into and out of methods. For a method call $v_1 = m(v_2)$ to $m(p)\{\ldots\}$, the access-path $v_2.\pi$ in the calling context is mapped to $p.\pi$ in the callee method. Dataflow symbols that flow out of a method call (via heap locations reachable from formal arguments, or return statements) are mapped back into the caller environment. Finally, information local to the caller that does not flow through the method call to m is propagated over the method call.[5] Legato's analysis is non-standard only in how values are transformed across method boundaries. Values that flow out of a method are transformed by the function ρ and values propagated over a method call are transformed by τ. We define these functions in this section. The full environment transformers are given in Figure 9.

4.1 Transitive Resource Accesses

In a language with methods, a single primed `get()` label is no longer sufficient to uniquely identify a resource access at some point in time. Consider the code sample in Figure 10. After line 5, the value in `b` comes from the most recent invocation of `get`^{1}`()`. However, after `m` is called again on line 6, the value in `b` comes from an execution of `get`^{1}`()` at some arbitrary point in the past. A single-primed label is unable to represent this situation. Leaving the value of $\widehat{1}$ in `b` after the second call to `m` is unsound, and using the $\bot$ value is imprecise. In general, transitive resource access may occur any arbitrary depth in the call-graph.

[5] For readers familiar with the IDE framework, these three components correspond to the call-to-start, exit-to-return-site, and call-to-return-site transformers respectively.

To precisely handle scenarios like the one in Figure 10, Legato generalizes the primed label ARV into *strings* of such labels. Unlike a single `get()` label, which identifies resource accesses relative to the current point in a programs execution, call-strings encode resource accesses relative to *other* program events: specifically, method invocations. For example, in the above example, the abstract resource version stored in `b` can be precisely identified by "the most recent invocation of `get`^{1}`()` that occurred during the most recent invocation of m at call site 2". The call-strings used as ARVs can precisely encode statements of this form.

A call-string takes the form $\widehat{k_1}^{p} \cdot \widehat{k_2}^{q} \cdots \widehat{k_m}^{r} \cdot \widehat{\ell}^{n}$, where $\widehat{\ell}$ is a primed `get()` label and each $\widehat{k_i}$ is a primed call site label. Call-strings are interpreted recursively; $s \cdot \widehat{k}^{n}$ represents the $(n+1)^{th}$ most recent invocation of $\mathtt{m}^{\mathtt{k}}()$ relative to the program point encoded in the prefix s. The string $s \cdot \widehat{\ell}^{n}$ has an analogous interpretation. If s is the empty string, the label is interpreted relative to the current point of execution. For example, the resource stored in `b` from Figure 10 can be represented by the ARV $\widehat{2} \cdot \widehat{1}$, which has the interpretation given above. As in the intraprocedural analysis, two distinct call-strings encode different invocations of a resource access, and thus their meet returns bottom. The lattice on call-strings is a constant, flat lattice on call-strings, which is a natural generalization of the lattice on individual labels.

When a value with call-string s flows out of a method m from the invocation $m^{k}()$, $\widehat{k}$ is prepended onto the string s. In other words, for a method call $v = m^{k}()$, $\rho \triangleq \lambda s. \widehat{k} \cdot s$. The prepended label encodes that the access represented by s occurs relative to the most recent invocation of m at k. Prepending $\widehat{k}$ also distinguishes transitive accesses that occurred while executing $m^{k}()$ from those resulting from other calls of `m()`.

The intraprocedural fragment of Legato remains primarily unchanged. Transitive resource accesses within a loop are handled with a priming mechanism similar to the one used for `get()` expressions. A string with $\widehat{k}$ at the head that is propagated over the method call $v = m^{k}()$ has a prime added to $\widehat{k}$. We define propagate over method transformer as:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot s' & \text{if } s \equiv \widehat{k}^{n} \cdot s' \\ s & o.w. \end{cases}$$

The justification for this transformation is identical to the one provided for values that flow over `get()` invocations. The added prime indicates any accesses that occurred relative to $\mathtt{m}^{\mathtt{k}}()$ now originate one more invocation of $\mathtt{m}^{\mathtt{k}}()$ in the past. Recursion is treated conservatively but does not require special handling in our analysis. Two iterations of the analysis through a recursive cycle will generate two strings, s and $s \cdot s$, the meet of which is $\bot$, ensuring fixpoint.

Soundness. We now informally argue for the soundness of the above approach. Recall from Section 3.4 that the soundness of Legato relies on a distinctness invariant, which states that if two values are derived from distinct resource accesses Legato must assign different abstract resource versions to those values. To simplify the following argument, we will assume that only a single value is returned from the callee via a `return` statement (the argument for values returned via the heap generalizes naturally from the following).

Let us assume the distinctness invariant holds for all values in the caller and callee environments, i.e., values from different invocations of `get()` are assigned different ARVs. Let us then show the invariant holds after the callee returns to the caller. First, it is immediate that the call-to-return transformer τ preserves distinctness for values in the caller environment. Next, suppose the value returned from the callee is derived from some resource access that occurred during the execution of the callee. To preserve the invariant, we must then show that the returned value is given a distinct ARV in the caller. By prepending the label of the call site and priming all ARVs that already contain that label, distinctness is ensured.

4.2 Dynamic Dispatch and Path-Sensitivity

Legato is not path-sensitive in general; as mentioned in Section 3.2 the abstract value of a variable from multiple branches are met at control-flow join points potentially yielding false positives. A key exception is Legato's handling of dynamic dispatch. In Java and other object-oriented languages, a method call m may dispatch to different implementations depending on the runtime type of the receiver object. In general, it is impossible to predict the precise runtime type of the receiver object for every call site, so a program's static call-graph has edges to every possible implementation $m_1, m_2, \ldots, m_n$ of `m` at the call site $\mathtt{m}^{\mathrm{k}}()$. If Legato treated multiple return flows like control-flow constructs such as `if` and `while`, the analysis would be sound but unacceptably imprecise.

Legato handles dynamic dispatch path-sensitively by aggregating results from each distinct concrete callee into a single, non-$\bot$ ARV. Although the resulting ARV encodes multiple, potentially incompatible resource accesses, Legato ensures that all accesses represented by the ARV come from different concrete callees of a single virtual call site. As only one concrete callee is invoked per execution of a virtual call site, only one access represented in an ARV may be realized at runtime. Thus, combining results from different concrete implementations into a single ARV does not allow for violations of at-most-once.

Multiple resource accesses are represented by generalizing the call-string representation from the previous subsection into tries, which encode *sets* of call-strings. Leaf nodes of the trie are labeled with primed `get()` labels, and interior nodes with primed call site labels. The children of a call site node labeled $\widehat{k}^n$ represent the possible results returned from the $(n+1)^{th}$ most recent invocation of the call site with label k. A path through the trie implicitly defines a call-string with the same interpretation as given in Section 4.1. The call-string representation of the previous subsection is a degenerate case of the trie representation where each node has only one child.

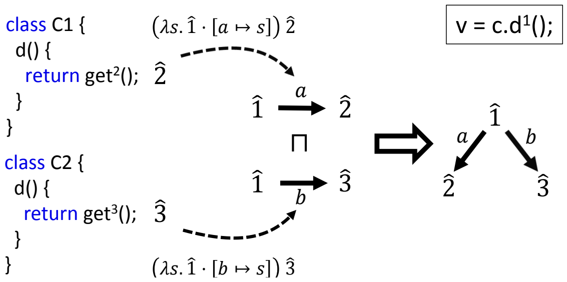
Formally, we write $\widehat{k}^n \cdot [b_1 \mapsto t_1, b_2 \mapsto t_2, \ldots]$ to represent a call site node $\widehat{k}^n$ with children $t_1, t_2, \ldots$ reachable along branches with ids $b_1, b_2, \ldots$ The branch ids are unique within each call site node and correspond to a potential callee. We call the branch id to child mapping the *branch map*, and write $\mathcal{M}$ to denote an arbitrary mapping.

We extend the return transformer ρ as follows. On return from a concrete implementation m_p to the call site $m^k()$, $\rho \triangleq \lambda s.\widehat{k} \cdot [p \mapsto s]$. That is, the ARV s is extended with a new call site root node labeled $\widehat{k}$ that has a single child with branch id p. In the caller, these single-child ARVs are aggregated into a single node that represents all possible results from each callee. Similarly, we update the function τ as follows:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot \mathcal{M} & \text{if } s \equiv \widehat{k}^n \cdot \mathcal{M} \\ s & \text{o.w.} \end{cases}$$

Combining ARVs from *different* invocations of the same virtual call site or different call sites yields $\bot$. To combine ARVs representing results from the *same* invocation of a call site, the branch maps of the ARVs are met pointwise by branch id. As is standard, unmapped branch ids in either map are assumed to have the value $\top$. However, if the meet of any branch is $\bot$ then the entire meet operator yields $\bot$. That is, a violation of at-most-once in one possible callee yields an overall inconsistent result. An example return flow and meet is shown in Figure 11. Formally, the full meet operator for trie ARVs is as follows:

$$\widehat{i}^n \cdot \mathcal{M}_1 \sqcap \widehat{j}^p \cdot \mathcal{M}_2 = \begin{cases} \widehat{i}^n \cdot \mathcal{M}' & \text{if } i = j \wedge n = p \wedge \mathcal{M}' \neq \bot \text{ where } \mathcal{M}' \triangleq \mathcal{M}_1 \sqcap \mathcal{M}_2 \\ \bot & o.w. \end{cases}$$



```
class C1 {
  d() {
    return get²();
  }
}
class C2 {
  d() {
    return get³();
  }
}
```

$(\lambda s.\widehat{1}\cdot[a\mapsto s])\,\widehat{2}$
$\widehat{2}$
$\widehat{1} \xrightarrow{a} \widehat{2}$
$\sqcap$
$\widehat{1} \xrightarrow{b} \widehat{3}$
$\widehat{3}$
$(\lambda s.\widehat{1}\cdot[b\mapsto s])\,\widehat{3}$
v = c.d¹();
$\widehat{1}$
a
b
$\widehat{2}$
$\widehat{3}$

Figure 11 Example of LEGATO's handling of dynamic dispatch. `v = c.d`^{1}`()` may dispatch to either implementation in `C1` or `C2`. The dashed lines illustrate the return flows, and are annotated with the return flow function applied by the analysis. The two single-child ARVs are met to produce the trie on the right. `a` and `b` are the branch ids assigned to the callees `C1.d` and `C2.d` respectively.

4.3 Effectively Identity Flows

Prepending labeled nodes on *all* return flows can cause imprecision. For example, consider:

```
idA(i) { return i }
idB(j) { return j }
x = get¹();
y = id²(x)
```

where `id` may dispatch to one of `idA` or `idB`. In this example, `x` is assigned $\widehat{1}$ and `y` is assigned $\widehat{2}\cdot[a\mapsto\widehat{1}, b\mapsto\widehat{1}]$. According to the lattice, these two values are distinct and may not be safely combined, despite being identical. This issue arises because the invocation of `id` is unnecessary to identify the resource access that flows to `y`, nor does the behavior of the two possible callees of `id` differ. We call a scenario like the above an *effectively identity flow*.

LEGATO handles effectively identity flows by detecting when the standard meet operator would produce $\bot$, and refining the ARVs to eliminate any effectively identity flows. Call-site nodes are added on return from a method invocation $m()$ to either identify transitive resources accesses (Section 4.1) or to differentiate behavior of multiple callees at $m()$ (Section 4.2). Conversely, if all callees exhibit the same behavior and no transitive resource accesses occur within the call $m()$, call site nodes added on return flow from $m()$ are, by definition, redundant. LEGATO *cannot* add labels on return only when necessary to disambiguate different resource accesses. Such an approach would require non-distributive environment transformers, which are unsuitable for use with the IDE framework upon which LEGATO is built.

Based on this intuitive definition of effectively identity flows, we define a refinement operation $\mathcal{R}$, which traverses the ARV trie, and iteratively removes redundant nodes. After the operation is complete, only the nodes and corresponding labels necessary to either distinguish a resource access or differentiate multiple callees' behavior are left in the trie. We first formally define effectively identity flows (EIF) and initial refinement operation $\mathcal{R}_0$ for the single dispatch case (Section 4.3.1). The definitions of EIFs and the full refinement operation, $\mathcal{R}$, for dynamic dispatch (Section 4.3.2) build upon these definitions.

4.3.1 Effectively Identity Flows and Single Dispatch

As a simplification, we consider call-strings with no primes: the operations and sets defined here can be easily extended to ignore primes on call labels. For every method m, let $\mathcal{AS}(m)$ denote the set of transitively reachable, unprimed, call site and `get()` labels of m. Further, for each call site label k we denote the method invoked at k as $\mathcal{CS}_k$. A call-string s contains an EIF if there exists a suffix $\widehat{k} \cdot s'$ such that there exists a $\widehat{j}$ in s' such that $j \notin \mathcal{AS}(\mathcal{CS}_k)$. The existence of $\widehat{j}$ indicates that the ARV must have been returned out of some method other than those called by $\mathcal{CS}_k$, and, by definition, the access represented by the ARV must therefore have occurred in some method other than those called by $\widehat{k}$. Thus, $\widehat{k}$ is irrelevant for the purposes of identifying the resource access encoded in the ARV.

The initial refinement operation, $\mathcal{R}_0$, follows from this definition. Let s be a call-string, $\widehat{k}$ the first label in s involved in an effectively identity flow, and $\widehat{j}$ be defined as above. Finally, let s'' be the suffix of s that starts with $\widehat{j}$ (inclusive). Given these definitions: $\mathcal{R}_0(s) \triangleq \mathcal{R}_0(s'')$. The refinement operation is defined inductively: in the base case where s contains no identity flows the refinement operation is defined to be $\mathcal{R}_0(s) \triangleq s$. Intuitively, the refinement operation iteratively strips off substrings of labels that form effectively identity flows until reaching the suffix of labels that are necessary to distinguish the resource access.

4.3.2 Effectively Identity Flows and Path-Sensitivity

In the presence of ARV branching, we must extend the definition of effectively identity flows presented above. In the single-dispatch case, call site nodes were necessary only to precisely represent transitive accesses; nodes that did not fulfill this purpose could be removed. In the presence of branching, a call site node may also be required to precisely combine otherwise incompatible method call results. Thus, a call site node is part of an effectively identity flow *iff* it is not required to identify accesses within a method call (as before) *and* it does not differentiate two or more otherwise incompatible method results.

We define an effectively identity flow in the presence of branching as follows. Each node encodes a finite set of strings, with each string corresponding to labels on a path from the node to the leaves of the ARV trie. Passing through a call site node $\widehat{i}$ along branch b corresponds to $\widehat{i_b}$. We will denote the set of call-strings for a node n with $n^{\natural}$. Similarly a call-string ARV can be trivially converted into a trie ARV as follows:

$$[\![\widehat{i_b} \cdot s]\!] \triangleq \widehat{i} \cdot [b \mapsto [\![s]\!]] \qquad [\![\widehat{i}]\!] \triangleq \widehat{i}$$

Given these definitions, an ARV contains an effectively identity flow if there exists a call site node $n \equiv \widehat{k} \cdot \mathcal{M}$ that satisfies two conditions. First, every call-string $\widehat{k_b} \cdot s \in n^{\natural}$ contains an effectively identity flow according to the definition in Section 4.3.1 originating at k. In other words, the call site node $\widehat{k}$ is unnecessary to identify any resource accesses within the call at k. The second condition is $\bigsqcap_{s \in n^{\natural}} [\![\mathcal{R}_0(s)]\!] \neq \bot$. That is, after removing the call site node $\widehat{k}$, it must be possible to meet the resulting ARVs without producing a violation of at-most-once. For nodes that satisfy this condition, the full refinement operation is: $\mathcal{R}(n) \triangleq \mathcal{R}(\bigsqcap_{s \in n^{\natural}} [\![\mathcal{R}_0(s)]\!])$. The base case for nodes that cannot be refined is $\mathcal{R}(n) \triangleq n$. Similarly to the single-dispatch case, the refinement operation traverses the ARV trie, stripping redundant nodes and collapsing redundant branches.

4.4 Application Level Concurrency

The at-most-once condition obviates reasoning about concurrent resource updates, but Legato must still account for concurrency within an application. Legato is not sound in the presence of data races: we assume that all mutable, shared state is accessed within a lock protected region. Thus, outside of synchronized regions, each thread reads only values previously written by that thread. However, within a synchronization region, a thread may observe values written by *any* other thread. Legato conservatively assigns heap locations read in synchronization regions the abstract version $\widehat{\ell}$, where ℓ is a fresh, distinct label.

In other words, synchronization primitives *havoc* the abstract resource versions potentially shared among threads.

5 Implementation and Challenges

We implemented Legato as a prototype tool for Java programs. We used the Soot framework [47] for parsing bytecode and call-graph construction. We built the Legato analysis on an extended version of the Heros framework [7]. Although we state our analysis in terms of access paths for simplicity of presentation, we actually operate on *access graphs* [21] a generalization of access paths. Access paths can only represent heap locations accessible via a finite number of field references. In contrast, access graphs compactly encode a potentially infinite set of paths through the heap. The analysis presented here extends naturally from access paths to access graphs.

To resolve uses of the Java reflection API, we relied on the heuristics present in the underlying Soot framework. However, we also found all of the applications in our evaluation suite provided a mechanisms for one method to invoke another based on an application-specific URL recorded in a static configuration file. We found that, like many uses of Java reflection [46, 2], these mechanisms are almost always used with static strings. Following the technique outlined in [46], where possible we use these strings to statically resolve these implicit calls to a direct call to a single method. When these heuristics fail, we soundly resolve to all possible callees. Unlike the Java reflection API, which must consider all methods/constructors as possible targets, the set of potential callees was small enough that this over-approximate approach was feasible in practice.

We do not include the full Java Class Library (JCL) in our analysis for performance reasons. This exclusion is only a source of imprecision in our analysis. For certain methods (e.g., members of the collections framework) we provide highly precise summaries. For unsummarized methods, Legato conservatively propagates information from arguments to return values/receiver objects similar to TaintDroid [16].

5.1 Alias Queries

To resolve the *mayAlias* queries on heap writes (see Section 3.5), we use a demand-driven, context and flow-sensitive alias resolution [41]. A single alias query must complete within a user-configurable time limit; if this budget is exceeded, Legato reports the configuration value as lost into the heap similar to the approach taken by Torlak and Chandra [44]. This was not a source of any false positives in our evaluation. We take a similar approach on flows of resources into static fields. Static fields are global references that persist throughout the entire lifetime of the program. We conservatively flag any write of a resource derived value that flows into a static field. This dramatically improved our alias resolution time and did not lead to many false positives.

5.2 Resource Model

The analysis described in Sections 3 and 4 is stated in terms of only one external resource. Our implementation handles multiple resources by operating over maps from resource names to individual ARVs. For generality, our implementation is parameterized over the resource access model of an application. A model defines the resource access sites in an application, and for each site returns the set of resource names potentially accessed at that site. The soundness and precision of Legato depends on the choice resource model: a model that omits some access sites may cause Legato to miss potential bugs. Similarly, an overly coarse model will be sound but likely imprecise in practice. However, in our evaluation we found that resource access sites are easy to identify in practice; we describe the resource models used in for our evaluation in Section 6.

The resource model used with Legato is unconstrained in the choice of resource names. This flexibility enables the use of an imprecise model when resources may alias, or when the exact name of resources cannot be determined precisely at analysis time. Under an imprecise resource model, all access sites that may access the same concrete external resource are mapped to a common abstract resource name. For example, all accesses to files with the extension `.txt` may be mapped to the logical resource name `*.txt`. A similar approach may be used when two or more resources interact or share state, i.e., resources with distinct names that share state may be given the same abstract resource name.

5.3 Context-Sensitivity

Each call site of a method m may call the method with different abstract input values. However, the IDE framework computes the values within m by taking the meet over all abstract inputs. This leads to imprecision in the following scenario:

```
do_print(a) {
  print(a);
}
do_print(get^1());
do_print(get^2());
```

The standard value computation within `do_print` would assign `a` the value $\widehat{1} \sqcap \widehat{2} = \bot$ which is imprecise. Initial versions of Legato used the context-insensitive value computation provided in Heros [7], but our results were impractically imprecise.

To overcome this imprecision, the Legato implementation extends the value computation phase of Heros to make it context-sensitive. We require an initial context and a context extension operator. At a call site to method m, the context of the call site C is extended with the extension operator, yielding the context C' for values computed within m originating from context C. The original value computation pass of the IDE framework is then executed for the method body with respect to the new context.

In our instantiation, we use an adaptive, k-limited call-string context scheme similar to that in [35]. To trade-off precision and scalability, we initially run the value analysis pass with all contexts limited to length 1. If Legato derives the value $\bot$ for some method parameter p in context C, it consults the corresponding argument values in all incoming contexts. If the argument in each incoming contexts is non-$\bot$, Legato infers that the $\bot$ value computed for p was due to insufficient context-sensitivity. Legato then adaptively increases the context-sensitivity for all such call sites, and then re-runs the value computation phase. This process is repeated until no $\bot$ parameter values arise due to insufficient context-sensitivity, although we impose an configurable artificial maximum length (6 in our experiments) to ensure termination. In our experiments, this limiting was the source of only 3 false positives.

The approach described above is necessarily more expensive than the original IDE framework, which runs only one value computation phase. In practice, the context-sensitive value computation phase does not significantly contribute to analysis time for two reasons. First, Legato needs only a handful of value computation phases to either rule out false positives from insufficient context-sensitivity or reach the configured limit. Second, within each value computation phase, values are computed within a method using context-*insensitive* summary functions, which are generated in an initial pass of the IDE analysis. These summary

Table 1 Measures of application complexity in the evaluation suite. **# IR Statements** is the count across all methods of statements in the intermediate representation used by Soot.

Program	Classes	Methods	Call Graph Edges	# IR Statements	Options
snipsnap	643	3,318	20,079	68,841	19
vqwiki	506	5,019	43,211	145,891	73
jforum	528	3,075	15,607	41,319	48
subsonic	886	4,578	20,768	67,615	44
mvnforum	938	10,548	132,712	409,847	90
personalblog	371	1,427	8,186	25,514	16
ginp	205	1,011	8,100	26,448	7
pebble	576	2,989	20,646	66,477	7
roller	853	4,735	30,229	95,439	29
blojsom	471	1,782	15,846	26,786	67

functions are symbolic abstractions of the method behavior on all possible input values. As a result, there is no need to re-analyze a method under each new context, which keeps recomputing values under new contexts relatively inexpensive.

5.4 Limitations

A fundamental limitation of our analysis is that we do not consider any possible synchronization between resource updates and resource accesses or between multiple resource accesses. This limitation will only yield false positives, as this means our analysis may be overly conservative in considering a program's resource accesses. Our prototype could, with modest effort, include annotations to indicate an access always returns the same abstract version or multiple access sites return the same abstract version.

Our analysis soundness is stated only in terms of direct information flow, i.e., we ignore the effects of implicit flow. Thus, LEGATO will fail to detect when two or more accesses of the same resource indirectly flow to a program value. We experimented with a version of the analysis that considered implicit flow but, as is common [22], the ratio of false positives to true positives was overwhelming.

As mentioned above, LEGATO relies on the Soot analysis framework for call-graph construction, reflection resolution, type hierarchy construction, etc. Thus, LEGATO is sound modulo the soundness of the underlying Soot framework implementation.

6 Evaluation

To evaluate LEGATO, we focused on the issue of consistency in the presence of dynamic configuration updates. A dynamic configuration update (DCU) is a configuration change that occurs at run time that takes effect without program restart. We chose this problem as representative of the broader problem of consistent dynamic resource usage, as we are unaware of any existing static analysis that is capable of effectively addressing this problem. The only tool we are aware of in this area is our prior work on Staccato [43], which is a dynamic analysis that may yield false negatives. In addition, Staccato LEGATO is parameterized over the dynamic resource being analyzed.

We are interested in the following questions:

- Does Legato find dynamic resource consistency errors in the analyzed applications with a reasonable ratio of true to false positives?
- Are the time and memory requirements to run Legato reasonable?

Experimental Setup. We evaluated Legato on 10 Java server applications. A summary of the applications and metrics related to code base and call graph size (as measures of application complexity) can be found in Table 1. We selected these applications from three sources. Subsonic and JForum come from our prior work on Staccato we include them for comparison with prior results.[6] Personalblog, Snipsnap, Roller, and Pebble are from the Stanford SecuriBench suite [28], a set of commonly analyzed web apps [29, 23].[7] Finally, we also used applications from prior work by Tripp et al. on TAJ [46], a taint analysis for web applications. We used all projects from TAJ's evaluation that satisfied the following conditions: a) the source code is publicly available, b) the project is a single, self-contained application, and c) the application supports dynamic configuration updates. The applications satisfying these conditions are VQWiki, MVNForum, Ginp, and Blojsom. Where possible, we used the same versions of the projects as those used in the original TAJ paper.

The dynamically configurable options of every application may be changed by an administrator at any point while processing a request. Across all our applications, applications accessed the configuration by reading from a global, in-memory map. When the configuration is changed by an administrator (either via the web interface or editing the on-disk configuration file) a thread in the application updates the in-memory configuration map. This thread runs concurrently with request handler threads that read from the configuration map.

Given this implementation pattern, we treated each individual option as a separate resource that can change at any moment. Every application accessed configuration options by either passing static strings to a key-value API (e.g., `Config.getValue("db-password")`) or calling option-specific getter methods (e.g., `Config.getDBPassword()`). We implemented generic resource models for these two access patterns. When analyzing an application, we specialized the appropriate model with an application-specific configuration YAML file which described the application's configuration API. The longest such file was only 195 lines. The number of options tracked for each application are included in Table 1.

All of the applications in our evaluation were written to run in a Java Servlet container [32]. To soundly model these applications, we generated driver programs based on the servlet container specification and used sound stub implementations of the servlet API. For heavily used parts of the Java Class Library, such as the collection and database APIs, we used hand written summaries. For other methods without implementations, we used the over-approximation of method behavior discussed in Section 5.

We performed two experiments. To measure the effectiveness of Legato, we ran the analysis on each evaluation program, and recorded all at-most-once violations reported by the analysis. We then manually classified these reports as either a true bug or false positive. (Where possible, we reported any true bugs we found to the original developers.)

To measure the performance of Legato, we ran the analysis 5 times for each application while collecting timing and memory usage information. We break down the time of the analysis into three components: call-graph construction time, alias query resolution time,

[6] Staccato was also applied to Openfire, but was used only to detect out-of-date configurations, an orthogonal issue to consistency.
[7] The SecuriBench suite contains 9 applications, but the remaining 5 do not support DCU.

Table 2 Bug reports from LEGATO. **TP** and **FP** are the numbers of true and false positives respectively. The last four columns record sources of false positives: **PS** is path-insensitivity, **SYN** is the conservative handling of synchronization, and **SF** is the conservative handling of static fields discussed in Section 5. **O** counts causes not included in the above categories, and includes imprecision due lack of application-, library-, or framework-knowledge. t\o indicates no reports due to timeout.

Project	TP	FP	PS	SYN	SF	O
jforum	4	14	2	1	3	8
ginp	7	1	0	0	1	0
vqwiki	12	8	2	4	1	1
snipsnap	2	2	1	0	1	0
pebble	0	4	3	0	0	1
subsonic	31	12	1	9	2	0
personalblog	1	3	3	0	0	0
roller	6	5	1	0	1	3
mvnforum	2	27	19	0	0	8
blojsom	t\o	t\o	t\o	t\o	t\o	t\o

and core analysis time, and report the average of these times. To measure the memory requirements of LEGATO, we sampled the heap size of the JVM every second. We intentionally avoid garbage collection before sampling the heap size. We found that excessive garbage collection caused an artificially high number of alias query timeouts, which ultimately skewed the analysis results and reported memory requirements.

All experiments were run on AWS EC2 m4.xlarge instances with 4 virtual CPUs at 2.4GHz, using the OpenJDK VM version 1.7.0_131, with 10GB of memory allocated to the JVM. We limited all aliasing queries to ten seconds, and set a 15 minute timeout for each run of the analysis.

6.1 Analysis Effectiveness

The results of running LEGATO on programs in our evaluation suite are shown in Table 2. LEGATO successfully completed within the 15 minute budget on 9 of the 10 applications in our evaluation suite (we discuss the reason for Blojsom's timeout below). Of the 9 applications on which LEGATO completed, the analysis found bugs in 8. Although the false positive *ratio* is relatively high, we were able to classify the results with minimal effort as many of the false positives were obvious. In many cases (84.6% of column **PS**) LEGATO detected that it lost precision due to control-flow join and automatically flagged the result as a potential false positive. We also exploited that ARVs are traces of flows from access to report sites to help interpret errors reported by our tool. We were able to find these bugs with a simple resource model (Section 5.2) and without being experts in the programs.

There are potentially two sources of false positives: imprecision in the analysis and the at-most-once condition being too strong for application specific reasons. In practice, we found that all false positives were the result of imprecision in the analysis. The primary source of imprecision was the lack of general path-sensitivity in the analysis (column **PS**). For example, almost all of the path-sensitivity false positives in MVNForum (16) were the result of identical code being cloned across different branches of conditional statements. The second largest source of false positives was the conservative handling of code that required application-, library-, or framework-specific domain knowledge to precisely model (included in column **O**). For example, 8 false positives in the **O** column of JForum are due to imprecise

```java
// Instance (1)
request.setAttribute("url", config.getUrl());
request.setAttribute("baseurl", config.getUrl());
// Instance (2)
String url = "/space/" + encode(snip.getName());
url += "/" + encode(att.getName()));
// ...
String encode(String toEncode) {
  String encodedSpace = config.getEncodedSpace();
  return toEncode.replace(" ", encodedSpace);
}
```

Figure 12 Two simplified examples of the “double read” pattern found in Snipsnap.

```java
List<String> getPodcastUrls() {
  List<String> toReturn = new List<>();
  for(...) {
    String baseUrl = // ...
    int port = config.getStreamPort();
    toReturn.add(rewriteWithPort(baseUrl, port));
  }
  return toReturn;
}
```

Figure 13 A correlated access found in Subsonic, where the `"streamPort"` option is aggregated into the `toReturn` variable.

models of Java’s reflection API. Our control-flow graph contained a control-flow edge from the return-site of a `Method.invoke` reflective invocation to a `MethodNotFoundException` exception handler, when the represented control-flow path is actually unrealizable.

6.1.1 Sample Bugs

We now highlight some of the bugs found and discuss broad patterns we noticed in our results. Many bugs arose from three patterns: 1) two sequential accesses to the same configuration option, 2) using a configuration option in a loop, and 3) storing configuration derived data in a global cache that was not cleared on update.

Double Reads. We found 4 instances of applications immediately combining two successive reads of the same option. Two simplified instances we found in the Snipsnap program are shown in Figure 12. In the first instance, `config.getUrl()` returns a URL based on the dynamically configurable option specifying the location of the web application. If this option changes between the two accesses, the `request` object’s attributes will contain URLs pointing to two different locations. This could cause confusion for the user as only a subset of links on the page returned by Snipsnap would be valid.

The second instance is similar as the two invocations of `encode` both access the dynamically configured `encodedSpace` option. In this instance, the URL returned to the user will contain a mix of incorrectly and correctly encoded spaces. As with the first instance, this bug can cause links in the returned page to mysteriously fail to work.

The author of Snipsnap confirmed that these two instances corresponded to true bugs, but declined to fix them due to age of the project, lack of active deployments, and the author’s judgment that the bugs were not serious enough to warrant a fix [20].

Correlated Accesses within Loops. Out of the 65 true reports, 21 were instances of correlated accesses of configuration options within a loop. We counted instances where a value derived from a configuration option read within a loop is aggregated with configuration-derived values from previous iterations of the same loop. The aggregated value is therefore derived from multiple accesses of the same option, violating our at-most-once condition. The priming approach described in Section 3.3 was crucial to detect these bugs.

A simplified example of this pattern, found in Subsonic, is shown in Figure 13. The URLs computed by the method are used to generate an XML file served to podcast subscription clients. If some of the URLs generated by the method have inconsistent port numbers, the subscription client end-user would be presented with a handful of podcasts that fail to work. Further, unlike broken links on a webpage, the generated XML file is likely never seen by the end-user and thus it may not be obvious that a refresh may solve the problem.

We also found this pattern in other applications in our benchmark suite. For example, in MVNForum, a web forum application, the email module may send messages to multiple recipients, but constructs each message in different iterations of a loop. During each loop iteration, MVNForum reads configuration options that specify the message's sender name and address, which may yield a batch of messages with inconsistent sender information.

Finally, we found an example in VQWiki, a wiki web application, that potentially led to a corrupted search index. While constructing the index, VQWiki executes a loop to generate the set of documents to add to the index. Each loop iteration reads a configuration option that controls the location of the application's data files; this value is then stored in the indexed document. If the value of the option were to change between loop iterations, the index would be corrupted and only recover on the next complete index rebuild.

Caching in Static Fields. As explained in Section 5.1, to avoid expensive alias queries for static fields while retaining soundness, we issue a report for each static field to which resource-derived information flows. This rough heuristic identified 4 instances where the at-most-once condition was violated due to caching.

For example, JForum (another forum application) can replace tokens in user text with embedded images of emojis. The URLs for these emojis, as with all URLs generated by JForum, are computed based on the dynamically configurable location of the forum application. JForum lazily computes the URL for every available emoji, then caches the results in a static field. However, if the administrator changes the base location of the application, this cache is not cleared. As a result, all links and images post-update will use the new location except for the emojis, which will be broken. Refreshing the page will not fix this issue as it requires the administrator to manually clear the emoji URL cache or restart the application.

In another more serious example, we found an instance in Roller where the login component cached whether password encryption was enabled in a static field populated at startup. However, user administration actions (e.g., update user, create user, etc.) always read the most up-to-date version of this flag, and encrypt passwords as appropriate. Thus, after changing this flag, any new users created by the administrator would be unable to log in until the entire application was restarted.

Other Patterns. We found multiple cases where configuration derived values were stored into the heap in one method, and then later combined with another configuration derived value in another method. A minimized example of this pattern, found in Ginp, is shown in Figure 14. Like most of the web applications in our evaluation, Ginp uses Java Servlet

```
 1 // in doStartTag
 2 this.cols = horiz / Config.getThumbSize();
 3 this.rows = vert / Config.getThumbSize();
 4 // in doAfterBody
 5 if(count - start >= this.rows * this.cols)
 6   showPicture = false
 7 // in _jspService (autogenerated)
 8 int _j_0 = _jspx_getpictures.doStartTag();
 9 // 41 lines of auto-generated code
10 int _e = _jspx_getpictures.doAfterBody();
```

Figure 14 Inconsistency bug found in Ginp. Detecting this bug requires precisely modeling framework code, and handling flows through method calls and the heap.

Pages (JSP), a dialect of HTML which allows mixing arbitrary Java code and user defined tags (such as `<ginp:getpictures.../>`). At page rendering time, JSP pages are transpiled into Java code and compiled. User defined tags are transformed into a sequence of calls to programmer defined callbacks. However, programmers generally only interact with the JSP source code and do not see the intermediate code containing the callback invocations.

The bug found by Legato involved one such user-defined tag. In one callback (`doStartTag`, lines 2 and 3), the same configuration option is read twice and stored into two seemingly unrelated heap locations. However, in a second callback (`doAfterBody`, lines 5 and 6) these two values are incorrectly combined to decide a loop condition. Finding this bug required precisely tracing the two abstract resource versions interprocedurally through the heap.

In another example, found in JForum, an SMTP mail session is constructed using the value of the dynamically configured mail host and then stored into an object field. In another method, this session is used to construct a transport, again using the value of the mail host option. If the mail host option changes between these two calls, the transport may try to connect to a mail host different from that of the mail session, which could cause the mail sending process to fail. Further, we confirmed that if the mail sending process failed with an exception, the messages to be sent were dropped and never resent.

Finally, we found a bug in Subsonic that relied on the application level concurrency approach described in Section 4.4. In this instance, a web request would initiate an update of an in-memory list of remote clients. However, this list was protected by a synchronized block. Legato concluded that a configuration-derived value placed in the list could be mixed with other configuration-derived values that originated from other threads.

Comparison with Staccato. To validate the effectiveness of our analysis, we compared the bugs found by Legato with those found by Staccato. A direct comparison is impossible, as Staccato uses slightly different correctness conditions, unsound heuristics not present in Legato, and also detects different types of errors orthogonal to the at-most-once condition. However, the 4 bugs found by Staccato in JForum and Subsonic that correspond to our at-most-once condition were detected by Legato. This finding partially validates that the bugs found by Legato correspond to true DCU bugs.

6.2 Performance

The results of our performance experiments are shown in Figure 16. Of the 10 applications, 9 finished within the 15 minute time limit, and 6 took less than a minute. For all applications in our evaluation suite, the 10GB heap limit was sufficient: the smallest peak heap size we

```
1 Request req = ...;
2 Response resp = ...;
3 HashMap context = new HashMap();
4 for(Plugin p : plugins) {
5   p.process(req, resp, context);
6 }
7 sendResponse(resp);
```

Figure 15 Sketch of code pattern that caused Legato to time out while analyzing Blojsom.

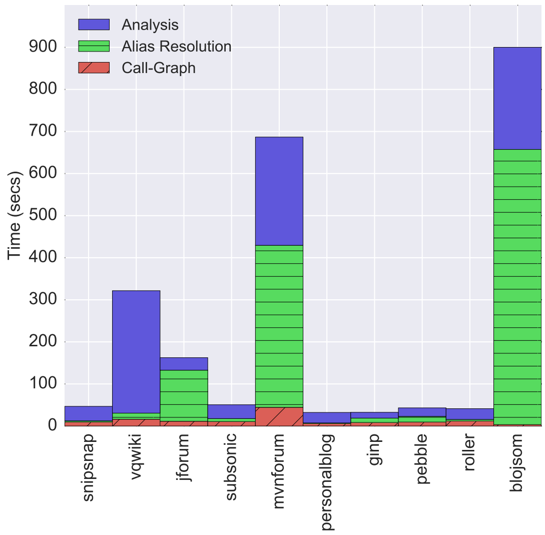


Figure 16 Analysis times for the evaluation targets.

observed was 0.5GB while analyzing Ginp and the largest was 7.5GB on MVNForum.

We now discuss the cause of Blojsom's timeout. The vast majority of Blojsom's 15 minute analysis budget was spent resolving alias queries. We found these expensive alias queries were caused by a problematic code pattern, which we sketch in Figure 15. Blojsom delegates the majority of request processing and application logic to 79 different plugins which are called via interface methods in a for loop during request processing (lines 4–6). To track per-request state, a shared HashMap `context` is also passed to each plugin; many plugins write configuration information into this map. To find all aliases of `context`, the alias resolver must explore all backwards paths of execution through the loop. Unfortunately, the megamorphic callsite on line 5 causes an explosion in the paths that must be explored, which quickly overwhelms the alias resolver. We could potentially address this issue by using a less precise approach to aliases, at the cost of overall analysis result quality.

7 Related Work

Typestate Analysis and Affine Type Systems. The phrase at-most-once often evokes linear (or more accurately, affine) type systems [49, 18, 45, 8, 13]. Both linear and affine type systems restrict how often a value may be used. Linear type systems guarantee that values may not be duplicated or destroyed, which enforces an exactly-once use discipline. Affine type systems allow destruction, which enforces an at-most-once use discipline. In contrast, under the at-most-once condition resources may be accessed multiple times, and may copied and re-used by the program. The at-most-once restriction only requires that each value depends only on at most one resource access.

Similar to linear and affine types, typestate analyses [42, 14, 50, 12, 17, 34] focus on verifying that the use of some object or resource follows a specific protocol. For example, the motivating example given in the original typestate paper by Strom et al. [42] is to verify that file handles are not written to after being closed. These access protocols are generally expressed in terms of an abstract state assigned to each object, and a set of methods or operations that cause transitions of object state according to some automaton.

The at-most-once condition is difficult to accurately capture using this framework. Although it would be possible to design an automaton to enforce that each resource was *used* exactly once during a value's computation, this condition is stricter than LEGATO's.

External Resources. There has been considerable effort into analyzing and understanding the external resources used by an application. For example, in the database community, recent work by Linares-Vásquez et al. [26] has looked at generating descriptions of how applications interact with databases. In a related work, Maule et al. among others [37, 30] have looked at evaluating the impact of database changes on applications. For configurable software understanding how software behaves under different configurations remains an active area of research [39, 24, 38]. Existing research on software configuration consistency has primarily focused on ensuring consistency between related configuration options. For example, Nadi et al. [33] examined constraints between compile-time configuration options for the Linux kernel. We consider this orthogonal to the consistency issues discussed here.

In addition to the above work on static external resources, verifying consistent behavior in the presence of *dynamic*, external resources has also been an active area of research. There has been considerable work in the security field to prevent vulnerabilities due to malicious, concurrent changes of the filesystem [36, 6, 31, 9].

Several decades of database research on transactions and isolation has focused on ensuring that applications interact consistently with the database. For example, serializeable isolation [5] can prevent check-then-act errors within a transaction by determining when a concurrent update has invalidated a previous read. Although this isolation can prevent consistency errors due to concurrent updates, empirical research performed by Bailis et al. [1] has shown that applications that eschew database level transactions (specifically Ruby on Rails applications) struggle to maintain consistency in the presence of concurrent writers.

To find errors in dynamic configuration update implementations (the instantiation considered in Section 5), our prior work on the Staccato dynamic analysis checks for correctness violations in applications with configuration changes at runtime [43]. One of the two correctness conditions checked by Staccato closely mirrors our at-most-once condition. However, Staccato does not consider multiple reads of the same option to be an error provided the same value is returned on each access. Thus, when considering multiple accesses on the same value, the at-most-once condition of LEGATO can be stricter than that checked by Staccato.

8 Conclusion

We presented LEGATO, a novel static analysis for detecting consistency violations in applications that use external resources. LEGATO verifies the at-most-once condition, which requires that all values depend on at most one access to each external resource. LEGATO efficiently checks this condition without explicitly modeling concurrency by using abstract resource versions. We demonstrated the effectiveness of this approach on 10 real-world Java applications that utilize dynamically changing configuration options.

References

1 Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

2 Paulo Barros, Suzanne Just, Renéand Millstein, Paul Vines, Werner Dietl, and Michael D Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *ASE*, 2015.

3 Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, 2014.

4 Arthur J Bernstein, Philip M Lewis, and Shiyong Lu. Semantic conditions for correctness at different isolation levels. In *Data Engineering*, 2000.

5 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.

6 Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing systems*, 2(2), 1996.

7 Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *State of the Art in Java Program analysis*, 2012.

8 Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

9 Xiang Cai, Rucha Lale, Xincheng Zhang, and Robert Johnson. Fixing races for good: Portable and reliable unix file-system race detection. In *Information, Computer and Communications Security*, 2015.

10 Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

11 Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In *TACAS*, 2006.

12 Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

13 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

14 Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, 2004.

15 Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, 1994.

16 William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 32(2), 2014.

17 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *TOSEM*, 17(2), 2008.

18 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1), 1987.

19 Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, 1979.

20 Matthias L. Jugel. Personal Communication, 2017.

21 Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *TOPLAS*, 30(1), 2007.

22 Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Information Systems Security*, 2008.

23 Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *ASE*, 2015.

24 Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *ASE*, 2014.

25 Yu Lin. *Automated refactoring for Java concurrency*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

26 Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *ISSTA*, 2016.

27 Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *OOPLSA*, 2014.

28 Ben Livshits. Standford securibench suite. `http://suif.stanford.edu/~livshits/securibench/`, 2017.

29 Benjamin Livshits. *Improving software security with precise static and runtime analysis.* PhD thesis, Stanford University, 2006.

30 Andy Maule, Wolfgang Emmerich, and David S Rosenblum. Impact analysis of database schema changes. In *ICSE*, 2008.

31 William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3), 1974.

32 Rajiv Mordani and Shing Wai Chan. Java servlet specification, 2009.

33 Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

34 Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, 2008.

35 Mayur Hiru Naik. *Effective Static Race Detection For Java.* PhD thesis, Stanford, 2008.

36 Mathias Payer and Thomas R. Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *VEE*, 2012.

37 Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *FSE*, 2013.

38 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.

39 Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

40 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), 1996.

41 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6116`, `doi:10.4230/LIPIcs.ECOOP.2016.22`.

42 Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12, 1986.

43 John Toman and Dan Grossman. Staccato: A Bug Finder for Dynamic Configuration Updates. In *ECOOP*, 2016.

44 Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *ICSE*, 2010.

45 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.

46 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.

47 Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, 2000.

48 David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

49 Philip Wadler. Linear types can change the world. In *IFIP TC*, 1990.

50 Daniel M Yellin and Robert E Strom. Protocol specifications and component adaptors. *TOPLAS*, 19(2), 1997.

A Appendix: Soundness

A.1 Preliminaries

Although the environment transformers presented in the main paper gave semantics as denotations from statements to environment transformers, the IDE framework of Sagiv et al. assigns transformers to edges in the program control flow graph. Following the notation of Sagiv et al. in [40], assume we have a function $M : E^* \to (Env \to Env)$, which maps an edge in the program control-flow graph to an environment transformer. This function naturally extends to paths of edges by composing the environment transformers for each successive edge in a path.

The solution computed by the IDE framework is the meet-over-all-paths solution,[8] defined for a distinguished start node s_0 and start environment Ω as:

$$MOP(n) \triangleq \bigsqcap_{p \in path(s_0, n)} M(p)(\Omega)$$

In other words, the meet-over-all-paths the meet of applying the transformers for every path from s_0 to n to the start environment Ω.

Our proofs exploit this path-based paradigm: we give the abstract and concrete instrumented semantics as assignments of transformers to edges. It is easy to see the correspondence to the transformers presented in the paper.

A.2 Concrete Instrumented Semantics

We first define the domain of concrete instrumented states as: $S = (X \to \mathbb{P}(\mathbb{N})) \times \mathbb{N}$, where X is the finite domain of variables that appear in a given program. We denote a concrete instrumented state of type S with $\langle env, c\rangle$. The instrumented semantics are given by the following assignment of transformers of type $S \to S$ to edges in the program supergraph:

$$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to s_1 \triangleq id \tag{1}$$

$$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to s_2 \triangleq id \tag{2}$$

$$\texttt{while}\ e\ \texttt{do}\ s\ \texttt{end} \to s \triangleq id \tag{3}$$

$$\texttt{while}\ e\ \texttt{do}\ s\ \texttt{end} \to s' \triangleq id \tag{4}$$

$$\texttt{skip} \to s \triangleq id \tag{5}$$

$$x = y \to s \triangleq \lambda\langle env, c\rangle.\langle env[x \mapsto env[y]], c\rangle \tag{6}$$

$$x = c \to s \triangleq \lambda\langle env, c\rangle.\langle env[x \mapsto \emptyset], c\rangle \tag{7}$$

$$x = y + z \to s \triangleq \lambda\langle env, c\rangle.\langle env[x \mapsto env[y] \cup env[z]], c\rangle \tag{8}$$

$$x = \texttt{get}^{\ell}\texttt{()} \to s \triangleq \lambda\langle env, c\rangle.\langle env[x \mapsto \{c\}], c + 1\rangle \tag{9}$$

[8] Technically, when considering interprocedural programs, the IDE framework computes the meet-over-all-*valid*-paths solution. As we do not consider methods in this section, we instead state our proofs using the simpler notion of meet-over-all-paths.

Where the edge in Equation (1) refers to the edge from the conditional header to the node corresponding to the branch statement s_1, and similarly for Equation (2) and the false branch s_2. The edge in Equation (3) corresponds to when the loop condition is true, and the loop body executed, whereas the edge in Equation (4) is when the loop condition is false and the loop is skipped. All other edges refer to the (unique) edge from a statement to its successor in the supergraph.

Define $C(p)$ as the composition of the transformers corresponding to each edge in the path p. Let Ω be the initial instrumented state, defined to be: $\langle \lambda\_.\emptyset, 0 \rangle$.

A.3 Abstract Semantics

Let the domain of primed labels presented in the paper be denoted by $L = \widehat{\ell}^n \cup \{\top, \bot\}$. The environments used in the paper are of type $\widehat{S} = X \rightarrow L$. We will denote environments of type $\widehat{S}$ with $\widehat{env}$. The distributive environment transformers in the paper are equivalent to the transformers of type $\widehat{S} \rightarrow \widehat{S}$ assigned to the edges in the supergraph:

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightarrow s_1 \triangleq id \tag{10}$$

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightarrow s_2 \triangleq id \tag{11}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \rightarrow s \triangleq id \tag{12}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \rightarrow s' \triangleq id \tag{13}$$

$$\texttt{skip} \rightarrow s \triangleq id \tag{14}$$

$$x = y \rightarrow s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \widehat{env}[y]] \tag{15}$$

$$x = c \rightarrow s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \top] \tag{16}$$

$$x = y + z \rightarrow s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \widehat{env}[y] \sqcap \widehat{env}[z]] \tag{17}$$

$$x = \texttt{get}^{\ell}() \rightarrow s \triangleq \lambda env.\lambda v. \begin{cases} \widehat{\ell} & \text{if } v \equiv v' \\ \widehat{\ell}^{n+1} & \text{if } env(v') \equiv \widehat{\ell}^n \\ env(v') & o.w. \end{cases} \tag{18}$$

Where the edges have the same interpretation as those given for the concrete semantics. At first glance, the use of id for loops and conditionals may appear incorrect. However, because the IDE framework computes the meet over all paths solution, the final result of the analysis takes the meet of all paths through a conditional, giving us the same effect. A similar observation applies for computing loop fixpoints.

Let $A(p)$ be the composition of the environment transformers corresponding to each edge in the path p, and let the initial abstract state Ω be defined to be $\top_{\widehat{S}}$, i.e., an environment that maps all variables to $\top$.

A.4 Proof

Define the invariant relation for two states as follows, $\langle env, c \rangle \sim \widehat{env}$ iff the following conditions hold:

$$\forall x.|env[x]| > 1 \Rightarrow \widehat{env}[x] = \bot \tag{Invariant 1}$$

$$\forall x, y, m, n. m \neq n \wedge env[x] = \{m\} \wedge env[y] = \{n\} \Rightarrow \widehat{env}[x] \neq \widehat{env}[y] \vee \widehat{env}[x] = \bot \vee \widehat{env}[y] = \bot \tag{Invariant 2}$$

$$\forall x.env[x] \neq \emptyset \Leftrightarrow \widehat{env}[x] \neq \top \tag{Invariant 3}$$

We now show that:

▶ **Theorem 1.**

$$\forall n, n', p, p'. p' \equiv p \circ n \circ n' \wedge p' \in \mathit{path}(s, n') \wedge C(p \circ n)(\Omega) \sim A(p \circ n)(\Omega) \Rightarrow C(p')(\Omega) \sim A(p')(\Omega)$$

Theorem 1 states that if the invariant holds for the two environments yielded by the transformers along the path $p \circ n$, the invariant still holds after applying the respective transformers for the edge $n \to n'$.

Proof. Let $C(p \circ n)(\Omega) = \langle \mathit{env}, c\rangle$ and $A(p \circ n)(\Omega) = \widehat{\mathit{env}}$, and let $C(p')(\Omega) = \langle \mathit{env}', c'\rangle$ and $A(p')(\Omega) = \widehat{\mathit{env}}'$. We assume $\langle \mathit{env}, c\rangle \sim \widehat{\mathit{env}}$ and must show that $\langle \mathit{env}', c'\rangle \sim \widehat{\mathit{env}}'$. We proceed on the type of edge $n \to n'$ that makes up the final component of the path p'.

Cases (1), (2), (3), (4), (5), (6), (7): Trivial

Case (8): It suffices to show that after executing the environment transformer all invariants hold for the variable x on the left-hand side of the assignment.

Invariant 1 If $|\mathit{env}[y]| > 1$ or $|\mathit{env}[z]| > 1$ then by definition of $\sim$, $\widehat{\mathit{env}}[y] = \bot$ or $\widehat{\mathit{env}}[z] = \bot$, and by the definition of meet, $\widehat{\mathit{env}}'[x] = \widehat{\mathit{env}}[y] \sqcap \widehat{\mathit{env}}[z] = \bot$, preserving the invariant. Consider the case now where $|\mathit{env}[y]| = 1 \wedge |\mathit{env}[z]| = 1 \wedge \mathit{env}[y] \neq \mathit{env}[z]$. Then by the definition of $\sim$, $\widehat{\mathit{env}}[y] \neq \widehat{\mathit{env}}[z]$ or one or both of $\widehat{\mathit{env}}[y]$ and $\widehat{\mathit{env}}[z]$ is $\bot$. In either case, $\widehat{\mathit{env}}'[x] = \widehat{\mathit{env}}[y] \sqcap \widehat{\mathit{env}}[z] = \bot$, again preserving the invariant.

Invariant 2 If $\mathit{env}'[x] = \{m\} = \mathit{env}[y] \cup \mathit{env}[z]$, then either:

1. $\mathit{env}[y] = \{m\}$ and $\mathit{env}[z] = \{m\}$. Then by invariant 3, we have that $\widehat{\mathit{env}}[y] \neq \top$ and $\widehat{\mathit{env}}[z] \neq \top$. If either $\widehat{\mathit{env}}[y]$ or $\widehat{\mathit{env}}[z]$ is $\bot$, then $\widehat{\mathit{env}}'[x] = \bot$ and the condition is trivially satisfied. Similarly, if $\widehat{\mathit{env}}[y]$ and $\widehat{\mathit{env}}[z]$ are distinct, non-$\bot$ values, then $\widehat{\mathit{env}}'[x]$ will be $\bot$ and again the invariant is trivially satisfied. Finally, consider the case where $\widehat{\mathit{env}}[y] = \widehat{\mathit{env}}[z]$. Then $\widehat{\mathit{env}}'[x] = \widehat{\mathit{env}}[y] = \widehat{\mathit{env}}[z]$, and thus the invariant must hold by transitivity of equality and the invariant relation on the input environments.
2. $\mathit{env}[y] = \{m\}$ and $\mathit{env}[z] = \emptyset$. Then invariant 3 implies that $\widehat{\mathit{env}}[y] \neq \top$ and $\widehat{\mathit{env}}[z] = \top$, whence $\widehat{\mathit{env}}'[x] = \widehat{\mathit{env}}[y]$. If $\widehat{\mathit{env}}[y] = \bot$ then the invariant is trivially satisfied, otherwise the invariant holds from the transitivity of equality and the invariant on the input environments.
3. $\mathit{env}[y] = \emptyset$ and $\mathit{env}[z] = \{m\}$ follows from symmetric reasoning to the above.

Invariant 3 If $\mathit{env}'[x] \neq \emptyset$, then $\mathit{env}[y] \neq \emptyset \vee \mathit{env}[z] \neq \emptyset$. By invariant 3 on the input environments, this implies that $\widehat{\mathit{env}}[y] \neq \top \vee \widehat{\mathit{env}}[z] \neq \top$. By the definition of meet, we must have $\widehat{\mathit{env}}'[x] \neq \top$ as required.

To establish the other direction of the bi-implication, it suffies to show that $\mathit{env}'[x] = \emptyset \Rightarrow \widehat{\mathit{env}}'[x] = \top$. If $\mathit{env}'[x] = \emptyset$, then $\mathit{env}[y] = \emptyset \wedge \mathit{env}[z] = \emptyset$, whence by the invariant on the input environments, we have $\widehat{\mathit{env}}[y] = \top \wedge \widehat{\mathit{env}}[y] = \top$. As $\top \sqcap \top = \top$, we have the desired result.

Case (9): We again establish the invariants post assignment.

Invariants 1 and 3 Trivial.

Invariant 2 By simple proof by contradiction, it can be shown that c is greater than any version number that appears in env. Thus, as $\mathit{env}'[x] = \{c\}$ is distinct from all other singleton version sets, it suffices to show that $\widehat{\mathit{env}}'[x]$ is likewise distinct from all other abstract versions. As the environment transformer in (18) adds a prime to existing values of the form $\widehat{\ell^n}$, this ensures that $\widehat{env}'[x] = \widehat{\ell}$ is unique within $\widehat{env}'$. Finally, for $y \neq x$, the priming process preserves inequality between abstract resource versions, ensuring the invariant holds. ◀

▶ **Corollary 2.** $\forall n, p.p \in path(s, n) \Rightarrow C(p)(\Omega) \sim A(p)(\Omega)$

Proof. By straightforward induction on path length and application of Theorem 1. ◀

We can now state the main soundness result:

▶ **Theorem 3.** $\forall p, n, x.p \in path(s, n) \wedge |C(p)(\Omega)[x]| > 1 \Rightarrow \left[\prod_{q \in path(s,n)} A(q)(\Omega)\right][x] = \bot$

In other words, Theorem 3 states that if any execution, at some point in the program a variable is derived from multiple versions of the resource, the analysis derives $\bot$ for that variable at that point.

Proof. Observe that if, for some x, $|C(p)(\Omega)[x]| > 1$, then $A(p)(\Omega)[x] = \bot$ by Theorem 2, and by the definition of meet, $\prod_{q \in path(s,n)} A(q)(\Omega)[x] = \bot$ ◀