Semantics-Aware Trace Analysis

Kevin Hoffman Patrick Eugster Suresh Jagannathan
Computer Science Department, Purdue University
{kjhoffma,peugster,suresh}@cs.purdue.edu

Abstract

As computer systems continue to become more powerful and complex, so do programs. High-level abstractions introduced to deal with complexity in large programs, while simplifying human reasoning, can often obfuscate salient program properties gleaned from automated source-level analysis through subtle (often non-local) interactions. Consequently, understanding the effects of program changes and whether these changes violate intended protocols become difficult to infer. Refactorings, and feature additions, modifications, or removals can introduce hard-to-catch bugs that often go undetected until many revisions later.

To address these issues, this paper presents a novel dynamic program analysis that builds a *semantic view* of program executions. These views reflect program abstractions and aspects; however, views are not simply projections of execution traces, but are linked to each other to capture semantic interactions among abstractions at different levels of granularity in a scalable manner.

We describe our approach in the context of Java and demonstrate its utility to improve *regression analysis*. We first formalize a subset of Java and a grammar for traces generated at program execution. We then introduce several types of views used to analyze regression bugs along with a novel, scalable technique for semantic differencing of traces from different versions of the same program. Benchmark results on large open-source Java programs demonstrate that semantic-aware trace differencing can identify precise and useful details about the underlying cause for a regression, even in programs that use reflection, multithreading, or dynamic code generation, features that typically confound other analysis techniques.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, diagnostics, testing tools, tracing

General Terms Algorithms, Reliability

1. Introduction

The ability to understand and analyze interactions among program components to infer or verify salient program properties is critical as software complexity increases.

This paper introduces a novel *semantics-aware* dynamic analysis for understanding and analyzing complex programs. Our approach achieves accuracy, flexibility and scalability through the concept of *semantic views*. Semantic views are trace abstractions which selectively aggregate collections of events with shared semantic traits found in a program execution trace, allowing for specific aspects and abstractions in programs to be captured accurately. Flexibility is achieved by allowing for new, specific views to be defined and applied. Views are linked among each other to capture program semantics in a scalable manner, and can be leveraged by profilers, optimizers, and bug-finders to quickly sift through a program execution, focusing attention on those parts that signify interesting deviations or properties.

We illustrate the usefulness of our approach by examining its utility to identify *regressions* in large (well-tested) software applications. We define a regression as a behavior that executed correctly under some input in a prior software version that is no longer correct under the same input in a newer version. Revisions are common in complex software systems, and refactorings and feature updates can introduce subtle regression bugs that are often not detected in a timely manner. Identifying and fixing these regressions can be a complex and time-consuming task, aggravated by the presence of advanced program mechanisms such as dynamic dispatch, code generation, and loading. For example, a query of the Apache Software Foundation bug tracking database[1] shows 455 bugs created during 2007 that involve a regression.[2] Out of these 455 bugs, 36% took longer than two months to resolve, 15% took longer than six months, and 11% remain unresolved as of November 2008.

Our technique provides a scalable solution to identifying precisely the cause of regressions in large, evolving Java programs. Given two traces corresponding to executions of a correct and regressing version of a program, we employ a novel differencing algorithm that extracts trace abstractions from these traces, and correlates executions based on their similarity by means of a novel, tractable, longest-common subseqeuence (LCS)-based approximation.

Motivating example. To illustrate the problem, Fig. 1 shows code snippets patterned after a known regression identified as `MYFACES-1130`[3] in the `Apache MyFaces` project,[4] an open source implementation of the `Java Server Faces` standard. The example centers around functionality in the `MyFaces` framework that automatically converts non-7-bit safe characters in the output of an HTTP request response into their equivalent HTML numeric entities. In this example, this conversion only occurs if the output document type is `text/html`, and each character is only converted

[1] `http://issues.apache.org/jira/`

[2] Bugs marked as duplicate or invalid were not included.

[3] `https://issues.apache.org/jira/browse/MYFACES-1130`

[4] `http://myfaces.apache.org/`

```java
public class ServletProcessor {
  NumericEntityUtil _binConv ;
  ...
  public void setRequestType(
      String rType, Logger log){
    if (rType.equals("text/html")){
      _binConv = new
          NumericEntityUtil(32, 127);
    }
    log.addMsg("Set request type to"
              +rType);
  }
  public StringBuffer processRequest(
                StringBuffer request)
                throws Exception
  {
    Servlet s = ...;
    StringBuffer resp =
         s.service(request);
    ...
    return _binConv.process(resp);
  }
}




...
```

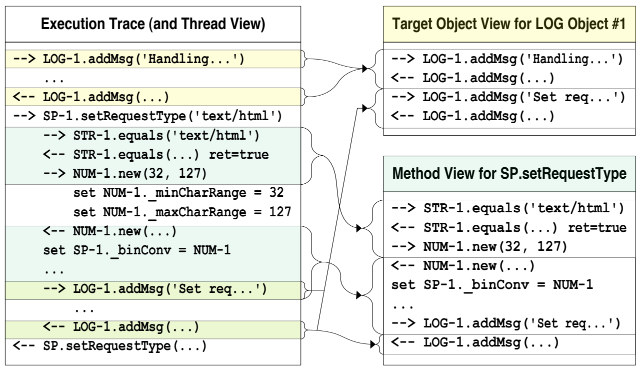
(a)

```java
public class ServletProcessor {

  ...
  public void setRequestType(
      String rType, Logger log){
    if (rType.equals("text/html")){
      addFilter(new
          BinaryCharFilter());
    }
    log.addMsg("Set request type to"
              +rType);
  }
  public StringBuffer processRequest(
                StringBuffer request)
                throws Exception
  {
    Servlet s = (Servlet)
         _servletType.newInstance();
    FilterChain chain = new
         FilterChain(_filters, 0, s);
    return chain.doFilter(request);
  }
}
class BinaryCharFilter ... {
  public BinaryCharFilter(){
    ... = new NumericEntityUtil(1,127);
  }
}
```

(b)

Figure 1. (a) an original non-regressing and (b) newer, regressing version of a program.



Execution Trace (and Thread View)
Target Object View for LOG Object #1
Method View for SP.setRequestType

```
--> LOG-1.addMsg('Handling...')
    ...
<-- LOG-1.addMsg(...)
--> SP-1.setRequestType('text/html')
    --> STR-1.equals('text/html')
    <-- STR-1.equals(...) ret=true
    --> NUM-1.new(32, 127)
        set NUM-1._minCharRange = 32
        set NUM-1._maxCharRange = 127
    <-- NUM-1.new(...)
    set SP-1._binConv = NUM-1
    ...
    --> LOG-1.addMsg('Set req...')
        ...
    <-- LOG-1.addMsg(...)
<-- SP.setRequestType(...)
```

```
--> LOG-1.addMsg('Handling...')
<-- LOG-1.addMsg(...)
--> LOG-1.addMsg('Set req...')
<-- LOG-1.addMsg(...)
```

```
--> STR-1.equals('text/html')
<-- STR-1.equals(...) ret=true
--> NUM-1.new(32, 127)
<-- NUM-1.new(...)
set SP-1._binConv = NUM-1
...
--> LOG-1.addMsg('Set req...')
<-- LOG-1.addMsg(...)
```

Figure 2. A portion of the execution trace for our example is shown on the left. The example is single threaded, so there is a single thread view which is identical to the full execution trace. There are many target object and method views. One target object view is shown for the first LOG object, containing only method calls and field accesses on that object. One method view is shown for the `SP.setRequestType` method, showing only actions that occurred when `SP.setRequestType` was on the top of the call stack.

if it is not in the range [32..127]. This range is defined programmatically and kept in mutable variables. A bug was introduced in a new version of this program that inadvertently sets the range to [1..127] instead of [32..127], causing a regression when given a document of type `text/html` that contains characters in the range [1..31].

In the original version, the `ServletProcessor` class directly instantiates the `NumericEntityUtil` with the correct range. This object is not used until after the HTTP request has been fully processed and the output generated; the actual character conversion process is affected by dynamic state initialized much earlier during program execution. In the new version, the `BinaryCharFilter` class was extracted from the `ServletProcessor` as part of a new generic I/O filtering abstraction, thus confounding static analysis as no readily apparent structural property is violated. The `BinaryCharFilter` class however provides an incorrect range of [1..127] to the new `NumericEntityUtil` object, causing the character filtering process to later produce incorrect results, but only for certain inputs.

This example is a pattern for an entire class of regressions caused when a piece of code incorrectly alters some dynamic state in the program, with the manifestation of the error appearing, only in certain cases, at some later point in the execution.[5] The causal distance between the point where the error occurs and where it manifests makes it difficult to precisely analyze such regressions manually, or correlate information derived from dynamic program slicing techniques that identify *all* semantically related operations to a regression.

Fig. 2 illustrates the intuition underlying our approach: different semantic views are used to capture individual aspects of the program and link them together. In the figure, we show views for the main thread of execution, as well as an object view for the `log` object, and a method view for method `setRequestType`. Observe that these specialized views record events that may be temporally far removed from one another. Our technique correlates these views with views generated by the regressing version to allow the regressing behavior to be localized and identified. Making such correlations precise and scalable is feasible because views discard actions unrelated to the behavior of the abstraction being traced.

[5] See also `https://issues.apache.org/jira/browse/SOAP-169`.

Contributions. This paper makes the following contributions:

1. **Semantic views:** Because execution traces can be millions of entries long, we define a new *view*-based trace abstraction that allow traces to be tractably analyzed and traversed. These views reflect the different levels of abstraction that naturally arise in object-oriented programs (e.g. objects, methods, threads, etc.), and provides a *semantics*-based means of structuring trace data. We present a formalization of traces and views for a subset of Java extending Featherweight Java with references, assignments, and threads. We furthermore define how these views can be correlated, linked, and navigated.
2. **Tractable trace differencing:** A new technique for differencing view-based traces is introduced, to correlate common points of execution by efficiently and accurately approximating the longest-common subsequence (LCS) [3] between full program traces. Notably, our differencing technique has linear complexity in both time and space, allowing full program traces that include dynamic state to be used in the analysis while keeping running times and memory requirements reasonable.
3. **Regression cause analysis:** We show how to use our differencing technique to accurately identify the causes of various regressions. The comparisons between traces for all difference sets are analyzed to produce a final candidate set of regression causes. Besides identifying differences that likely caused the regression, the analysis outputs a full semantic "diff" between the original and new versions, allowing these potential causes to be viewed in their full context, with dynamic state.

These techniques are implemented in a fully automated way in our tool, RPRISM, requiring no code annotations or access to source code. Benchmark results on realistic well-engineered Java programs demonstrate that our techniques yield high accuracy (few false positives or negatives produced), is scalable (traces with millions of entries can be analyzed), and is applicable even to programs that exploit advanced features such as multithreading, reflection, and dynamic code generation. To our knowledge, RPRISM is the first tool capable of performing automated regression analysis on programs with such a feature set.

The remainder of the paper is organized as follows: Sections 2 and 3 formalize our approach in the context of a subset of Java and view-based trace differencing respectively. Section 4 elaborates on these concepts in the context of regression analysis. An empirical evaluation of our techniques, including several large case studies, is presented Section 5. Section 6 describes related work; conclusions are given in Section 7.

2. View Model

In this section, we present (i) an abstract model of execution traces and a language whose evaluation produces such traces, and (ii) the *views* trace abstraction.

2.1 Language

We first introduce a simple object-oriented language whose syntax is shown in Fig. 3; its syntax and semantics follows in the spirit of other core object-based calculi such as Featherweight Java (FJ) [11] or Classic Java [7].

Our language augments FJ — modulo casts — with locations (of the form $l\texttt{(}C\texttt{)}$), field assignments ($t\texttt{.}f\texttt{=}t$), sequences of terms ($\overline{t}$), value objects ($\texttt{new}\ D\texttt{(}d\texttt{)}$), and threads ($\texttt{T(}\overline{t}\texttt{;)}$). A corresponding program is defined as such a thread term. Note that for brevity, locations $l\texttt{(}C\texttt{)}$ are sometimes simply written l when the type C is not germane to the context.

$$
\begin{array}{lrcl}
\mathit{program} & P & ::= & \texttt{T(}\overline{t}\texttt{;)} \\
\mathit{class} & CL & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{\{}\overline{A}\ \overline{f}\ \texttt{;}\ K\ \overline{M}\texttt{\}} \\
\mathit{creation} & K & ::= & C\texttt{(}\overline{A}\ \overline{f}\texttt{)\{super(}\overline{f}\texttt{);this.}\overline{f}\texttt{=}\overline{f}\texttt{;\}} \\
\mathit{method} & M & ::= & A\ m\texttt{(}\overline{A}\ \overline{x}\texttt{)}\ \texttt{\{}\overline{t}\texttt{;}\ \texttt{return}\ t\texttt{;\}} \\
\mathit{type} & A & ::= & C \mid D \\
\mathit{term} & t & ::= & x \mid v \mid t\texttt{.}f \mid t\texttt{.}f\texttt{=}t \mid t\texttt{.}m\texttt{(}\overline{t}\texttt{)} \mid \texttt{new}\ C\texttt{(}\overline{t}\texttt{)} \\
 & & \mid & \texttt{new}\ D\texttt{(}d\texttt{)} \mid \texttt{T(}\overline{t}\texttt{;)} \\
\mathit{value} & v & ::= & l\texttt{(}C\texttt{)} \mid D\texttt{(}d\texttt{)} \\
\mathit{primitive} & d & \in & \mathbb{D} \\
 & (D, d, \mathbb{D}) & \in & \{(\texttt{Bool}, b, \mathbb{B}), (\texttt{Int}, z, \mathbb{Z}), \ldots, (\texttt{Float}, r, \mathbb{R})\}
\end{array}
$$

Figure 3. Core language syntax.

2.2 Traces

Notably, program evaluation yields traces, whose structure is described below and outlined in Figure 4. Such a trace is a sequence of trace entries $\overline{\tau}{=}\tau^1\bullet\ldots\bullet\tau^n$, with $|\overline{\tau}|$ denoting its length (n). Different traces are identified by their names, shown in superscripts (e.g., $\overline{\tau}^L$). A trace entry $\texttt{entry(}eid, tid, m, \mu, e\texttt{)}$ is a five-tuple consisting of an entry identifier eid (the index of the entry in the trace), and four items forming a generic "context", namely: an identifier of the active thread (tid), the method under execution (m), a representation of the object on which m is executing (μ), and an event e that captures a specific action. For now an object is considered to be represented simply by its location l.

We make use of stacks to trace method calls. A stack $\mathcal{S}$ is a sequence of stack entries of the form $\texttt{s(}m,l,l'\texttt{)}$, representing the invocation of method m found in object l' from object l. We write $\mathcal{S}\bullet\texttt{s(}m,l,l'\texttt{)}$ for appending such an entry to a stack $\mathcal{S}$. We write $\overline{\mathcal{S}}$ to denote an ordered set of stacks; each element in this set captures the dynamic context of some actively executing thread. We write $\mathcal{S}_1 \cdot \ldots \cdot \mathcal{S}_n$ to enumerate the elements in this set. Similar notation is used to enumerate an ordered set of threads in a program state. As shorthand, we write $\overline{\mathcal{S}}\cdot\overline{\mathcal{S}'}$ to denote the ordered set of stacks derived by concatenation of the sets defined by $\overline{\mathcal{S}}$ and $\overline{\mathcal{S}'}$; similar shorthand is used to define concatenation over ordered sets of threads.

Fig. 5 presents relevant definitions. Besides the usual auxiliary functions for field and method body lookup, this includes evaluation contexts used in the definition of the language's operational semantics, and a function $\mathcal{E}\downarrow()$ for obtaining the representation of an object to be stored in a trace, revisited in the next section. For now, we ignore the representation of primitive values (thus, $\mathcal{E}{\downarrow}(D\texttt{(}d\texttt{)}){=}\_$), but assume that any object representation contains the original location, retrievable via $l{\downarrow}(\mu)$ for a given μ.

$$
\begin{array}{lrcl}
\mathit{event} & e & ::= & FE \mid ME \mid KE \mid TE \\
\mathit{field\,event} & FE & ::= & \texttt{get(}\mu, f, \mu\texttt{)} \mid \texttt{set(}\mu, f, \mu\texttt{)} \\
\mathit{method\,event} & ME & ::= & \texttt{call(}\mu, m, \overline{\mu}\texttt{)} \mid \texttt{return(}\mu, m, \overline{\mu}\texttt{)} \\
\mathit{object\,event} & KE & ::= & \texttt{init(}A, \overline{\mu}, \mu\texttt{)} \\
\mathit{thread\,event} & TE & ::= & \texttt{fork(}\overline{\mathcal{S}}\texttt{)} \mid \texttt{end(}\overline{\mathcal{S}}\texttt{)} \\
\mathit{object} & \mu & ::= & l \\
 & & & \\
\mathit{trace\,entry} & \tau & ::= & \texttt{entry(}eid, tid, m, \mu, e\texttt{)} \\
\mathit{entry\,ID} & eid & \in & \mathbb{Z} \\
\mathit{thread\,ID} & tid & \in & \mathbb{Z} \\
 & & & \\
\mathit{stack} & \mathcal{S} & ::= & \mathcal{S}_\emptyset \mid \mathcal{S}\bullet\texttt{s(}m, \mu, \mu\texttt{)}
\end{array}
$$

Figure 4. Trace syntax.

2.3 Dynamic Semantics

Fig. 6 defines an operational semantics for our language. Global evaluation is of the form $\langle\overline{t}, \overline{\tau}, \mathcal{E}, \overline{\mathcal{S}}\rangle \Longrightarrow \langle\overline{t}', \overline{\tau}\bullet\tau, \mathcal{E}', \overline{\mathcal{S}}'\rangle$ in which $\overline{t}$ is an ordered sequence of thread terms, $\overline{\tau}$ is the trace being generated, $\mathcal{E}$ is an object store, and $\overline{\mathcal{S}}$ is the ordered set of stacks corresponding to these threads. Local evaluation is of the form $\langle t, \overline{\tau}, \mathcal{E}, \mathcal{S}\rangle \underset{j}{\longrightarrow} \langle t', \overline{\tau}\bullet\tau, \mathcal{E}', \mathcal{S}'\rangle$ with j the index of the thread term t being reduced at that step. Rules for local evaluation only include one stack, $\mathcal{S}$, corresponding to the single thread term reduced.

Rule CONGR-E relates local and global evaluation. Rule FORK-E creates a new thread of control, and records a new entry in the trace whose event captures the stacks representing the newly created thread's parentage. Note that we track the creation context for the full ancestry of a thread (spawn-point call stack, call stack of spawn-point of spawning thread, etc.), in order to increase the accuracy of correlating threads between different traces. Rule END-E records the completion of a thread. Rule CONS-E defines object creation; the event associated with the object creation trace entry records the class name, the representation of the parameters to the constructor, and the representation of the created object itself. Rule CONS-VAL-E is defined similarly for primitives. Rules FIELD-ACC-E and FIELD-ASS-E record trace entries for field access and assignment. The definition of method call (rule METH-E) records a trace event that captures the object and method being invoked, along with its arguments (the calling context being captured by the enclosing entry). Similarly, rule RETURN-E records a trace entry that captures details relevant to a return action; this includes a trace event that records the method and object being returned from, and a representation of the return value.

2.4 *Views* Trace Abstraction

Although complete, the traces yielded by our evaluation rules require tedious interpretation to understand the underlying program behavior, since these traces capture features of higher-level constructs. We consequently formulate named projections over these low-level traces, which link *semantically related* trace events. These projections, termed *views*, represent various levels of abstraction in the executing program. Views are formed by mapping each trace entry to a set of *view names*, describing which specific views an entry is a member of. This set of names is obtained by creating a union of the results of all of the view name mapping

Definitions

$$fields(\texttt{Object})=\emptyset \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{\{}\overline{A}\ \overline{f}\ \texttt{;}\ \ldots\texttt{\}} \quad fields(C')=\overline{A}'\ \overline{f}'}{fields(C)=\overline{A}'\ \overline{f}',\ \overline{A}\ \overline{f}}$$

$$\frac{\texttt{class}\ C\ldots\ \texttt{\{}\ldots\ \overline{M}\texttt{\}} \quad A\ m\texttt{(}\overline{A}'\ \overline{x}\texttt{)}\ \texttt{\{}\overline{t}\texttt{;\}} \in \overline{M}}{mbody(m,C)=(\overline{x},\overline{t})} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{\{}\ldots\ \overline{M}\texttt{\}} \quad \nexists\ldots m\ldots \in \overline{M}}{mbody(m,C)=mbody(m,C')}$$

$$\mathcal{E}{\downarrow}(D\texttt{(}d\texttt{)})=\_ \qquad \mathcal{E}{\downarrow}(l\texttt{(}C\texttt{)})=l$$

Evaluation contexts

$$E ::= [] \mid E\texttt{.}f \mid E\texttt{.}f\texttt{=}t \mid v\texttt{.}f\texttt{=}E \mid E\texttt{.}m\texttt{(}\overline{t}\texttt{)} \mid v\texttt{.}m\texttt{(}E\texttt{)} \mid \texttt{new}\ C\texttt{(}E\texttt{)} \mid \overline{v}\texttt{,}E\texttt{,}\overline{t} \mid \overline{v}\texttt{;}E\texttt{;}\overline{t} \mid \texttt{T(}E\texttt{;)} \mid E\texttt{;return}\ t \mid \overline{v}\texttt{;return}\ E$$

Figure 5. Definitions and evaluation contexts.

functions (ν) (one mapping function is defined for each type). Specific views are then obtained by trace projections over view names of the form $\pi_{p_{\nu_\phi|\gamma}} : \overline{\mathbb{T}} \to \overline{\mathbb{T}}$ (see Figure 7), where $\overline{\mathbb{T}}$ is the domain of traces. $p_{\nu_\phi|\gamma}$ ranges over predicates of the form $\mathbb{T}\to\mathbb{B}$ with $\mathbb{T}$ the domain of trace entries. $p_{\nu_\phi|\gamma}$ models whether a given trace entry is a member of a specific view (as named by γ) of a particular view type (ϕ). We focus on four view types, which are defined by their view name mapping functions:

- **Thread views, ν_{TH}**: One thread view is defined for each thread tid executing in the program; it contains precisely those events that occur within that thread, in the order of execution.
- **Method views, ν_{CM}**: One method view is defined for each fully qualified method name m (for simplicity the class name is omitted in the figures) in the program. Each method view contains precisely those events that occur while that particular method was at the top of the call stack.
- **Target object views, ν_{TO}**: For each object a target object view is defined, containing precisely those events that occur when it *is the target of a method call or field access*.
- **Active object views, ν_{AO}**: For each object an active object view is defined containing those events that occur when it *is on the top of the call stack*.

The key to effective program analysis using our view model of tracing is that all these views are linked together. In the present context it is sufficient to view these links as being implicitly given by retaining indices of the original trace in the projected views. For example, a trace event recording a method call `o.m(...)` will be recorded in the thread view of the thread in which the call is performed, in a method view for `m`, in the active object view of the method performing the call, and in the target object view for target object `o`. The trace index found in the entry can be used to navigate from the entry found in one view to its position in another. In this way, a program as a whole may be modeled as a complex "web" of interconnected views. At any arbitrary point in any view, one can use these links to visit all semantically related views (as modeled by the defined view types), thereby organizing both the exploration of program execution as well as the presentation of the results of such an analysis in more meaningful ways.

3. Views-Based Trace Differencing

The comparison of traces through comparison of their corresponding view trace webs allows for a powerful program analysis. The foundation for such an analysis consists in the comparison of trace *pairs*. This section develops a semantics for "evaluating" pairs of such traces to identify semantic differences.

$$\boxed{\langle\overline{t},\overline{\tau},\mathcal{E},\overline{\mathcal{S}}\rangle \Longrightarrow \langle\overline{t}',\overline{\tau}\bullet\tau,\mathcal{E}',\overline{\mathcal{S}}'\rangle}$$

$$\frac{\langle t,\overline{\tau},\mathcal{E},\mathcal{S}_j\rangle \underset{j}{\longrightarrow} \langle t',\overline{\tau}',\mathcal{E}',\mathcal{S}_j'\rangle}{\begin{array}{c}\langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\texttt{T}_j'\texttt{(}E[t]\texttt{)}\cdot\overline{\texttt{T(}\ldots\texttt{)}}'',\overline{\tau},\mathcal{E},\overline{\mathcal{S}}\cdot\mathcal{S}_j\cdot\overline{\mathcal{S}}'\rangle\Longrightarrow\\ \langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\texttt{T}_j'\texttt{(}E[t']\texttt{)}\cdot\overline{\texttt{T(}\ldots\texttt{)}}'',\overline{\tau}',\mathcal{E}',\overline{\mathcal{S}}\cdot\mathcal{S}_j'\cdot\overline{\mathcal{S}}'\rangle\end{array}} \text{(CONGR-E)}$$

$$\boxed{\langle t,\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle t',\overline{\tau}\bullet\tau,\mathcal{E}',\mathcal{S}'\rangle}$$

$$\frac{\mathcal{S}_j\bullet\texttt{s(}m,\mu,\mu'\texttt{)} \in \overline{\mathcal{S}} \quad \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{fork(}\overline{\mathcal{S}}\texttt{))}}{\begin{array}{c}\langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\texttt{T}_j'\texttt{(}E[\texttt{T(}\overline{t}\texttt{;)}]\texttt{)}\cdot\overline{\texttt{T(}\ldots\texttt{)}}'',\overline{\tau},\mathcal{E},\overline{\mathcal{S}}\rangle\Longrightarrow\\ \langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\texttt{T}_j'\texttt{(}E[]\texttt{)}\bullet\overline{\texttt{T(}\ldots\texttt{)}}''\cdot\texttt{T(}\overline{t}\texttt{;)},\overline{\tau}\bullet\tau,\mathcal{E},\overline{\mathcal{S}}\cdot\mathcal{S}_\emptyset\rangle\end{array}} \text{(FORK-E)}$$

$$\frac{\overline{\mathcal{S}}=\overline{\mathcal{S}}'\cdot\mathcal{S}_j\bullet\texttt{s(}m,\mu,\mu'\texttt{)}\cdot\overline{\mathcal{S}}'' \quad \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{end(}\overline{\mathcal{S}}\texttt{))}}{\langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\texttt{T}_j'\texttt{(}\overline{v}\texttt{;)}\cdot\overline{\texttt{T(}\ldots\texttt{)}}'',\overline{\tau},\mathcal{E},\overline{\mathcal{S}}\rangle\Longrightarrow\langle\overline{\texttt{T(}\ldots\texttt{)}}\cdot\overline{\texttt{T(}\ldots\texttt{)}}'',\overline{\tau}\bullet\tau,\mathcal{E},\overline{\mathcal{S}}'\cdot\overline{\mathcal{S}}''\rangle} \text{(END-E)}$$

$$\frac{\begin{array}{c}l\notin dom(\mathcal{E}) \quad \mathcal{E}'=\{l\mapsto[f_1{:}v_1,\ldots,f_n{:}v_n]\}\mathcal{E} \quad \mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m,\mu,\mu'\texttt{)}\\ \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{init(}C,\overline{\mathcal{E}{\downarrow}(v)},\mathcal{E}{\downarrow}(l)\texttt{))} \quad fields(C)=\overline{A}\ \overline{f}\end{array}}{\langle\texttt{new}\ C\texttt{(}\overline{v}\texttt{)},\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle l\texttt{(}C\texttt{)},\overline{\tau}\bullet\tau,\mathcal{E}',\mathcal{S}\rangle} \text{(CONS-E)}$$

$$\frac{\mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m,\mu,\mu'\texttt{)} \quad \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{init(}D,\_,\mathcal{E}{\downarrow}(D\texttt{(}d\texttt{)})\texttt{))}}{\langle\texttt{new}\ D\texttt{(}d\texttt{)},\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle D\texttt{(}d\texttt{)},\overline{\tau}\bullet\tau,\mathcal{E}',\mathcal{S}\rangle} \text{(CONS-VAL-E)}$$

$$\frac{\begin{array}{c}\mathcal{E}(l)=[\ldots,f_i{:}v,\ldots] \quad \mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m,\mu,\mu'\texttt{)}\\ \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{get(}\mathcal{E}{\downarrow}(l),f_i,\mathcal{E}{\downarrow}(v)\texttt{))}\end{array}}{\langle l\texttt{.}f_i,\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle v,\overline{\tau}\bullet\tau,\mathcal{E},\mathcal{S}\rangle} \text{(FIELD-ACC-E)}$$

$$\frac{\begin{array}{c}\mathcal{E}(l)=[\ldots,f_i{:}v,\ldots] \quad \mathcal{E}'=\{l\mapsto[\ldots,f_i{:}v',\ldots]\}\mathcal{E} \quad \mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m,\mu,\mu'\texttt{)}\\ \tau=\texttt{entry(}|\overline{\tau}|,j,m,\mu',\texttt{set(}\mathcal{E}{\downarrow}(l),f_i,\mathcal{E}{\downarrow}(v')\texttt{))}\end{array}}{\langle l\texttt{.}f_i\texttt{=}v',\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle v',\overline{\tau}\bullet\tau,\mathcal{E}',\mathcal{S}\rangle} \text{(FIELD-ASS-E)}$$

$$\frac{\begin{array}{c}mbody(m,C)=(\overline{x},\overline{t})\\ \mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m',\mu,\mu'\texttt{)} \quad \mathcal{S}''=\mathcal{S}\bullet\texttt{s(}m,\mu',\mathcal{E}{\downarrow}(l)\texttt{)}\\ \tau=\texttt{entry(}|\overline{\tau}|,j,m',\mu',\texttt{call(}\mathcal{E}{\downarrow}(l),m,\overline{\mathcal{E}{\downarrow}(v)}\texttt{))}\end{array}}{\langle l\texttt{(}C\texttt{)}\texttt{.}m\texttt{(}\overline{v}\texttt{)},\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle\{{}^{l}\!/_{\texttt{this}},{}^{\overline{v}}\!/_{\overline{x}}\}\overline{t},\overline{\tau}\bullet\tau,\mathcal{E},\mathcal{S}''\rangle} \text{(METH-E)}$$

$$\frac{\begin{array}{c}\mathcal{S}=\mathcal{S}'\bullet\texttt{s(}m',\_,\_\texttt{)}\bullet\texttt{s(}m,\mu,\mu'\texttt{)}\\ \tau=\texttt{entry(}|\overline{\tau}|,j,m',\mu,\texttt{return(}\mu',m,\mathcal{E}{\downarrow}(v')\texttt{))}\end{array}}{\langle\overline{v}\texttt{;return}\ v',\overline{\tau},\mathcal{E},\mathcal{S}\rangle \underset{j}{\longrightarrow} \langle v',\overline{\tau}\bullet\tau,\mathcal{E},\mathcal{S}'\rangle} \text{(RETURN-E)}$$

Figure 6. Program evaluation.

3.1 Trace Differencing Semantics

We present a small-step semantics for "evaluating" pairs of traces ($\overline{\tau}^L,\overline{\tau}^R$) (termed the left and right traces) by comparing them in order to formalize the trace differencing process. The evaluation produces a set, Λ, containing those trace entries that are considered to be similar between the two traces. The set of differences is then easily computed from Λ and the original trace. We assume that before evaluation a special `eof` trace entry is appended to each trace, and is also appended as many times as needed to the shorter

$$\pi_p(\tau^1.\overline{\tau}) = \begin{cases} \tau^1.\pi_p(\overline{\tau}) & p(\tau^1) \\ \pi_p(\overline{\tau}) & \textit{otherwise} \end{cases}$$

$$p_{\nu_\phi|\gamma}(\tau) = (\gamma \neq \bot \wedge \gamma = \nu_\phi(\tau))$$

$$\nu_{\mathrm{TH}}(\texttt{entry}(i, j, m, \mu, e)) = \langle \mathrm{TH}, j \rangle$$

$$\nu_{\mathrm{CM}}(\texttt{entry}(i, j, m, \mu, e)) = \langle \mathrm{CM}, m \rangle$$

$$\nu_{\mathrm{TO}}(\texttt{entry}(i, j, m, \mu, e)) = \begin{cases} \langle \mathrm{TO}, l{\downarrow}(\mu') \rangle & e = \texttt{call}(\mu', m, \overline{\mu''}) \\ \langle \mathrm{TO}, l{\downarrow}(\mu') \rangle & e = \texttt{return}(\mu', m, \overline{\mu''}) \\ \langle \mathrm{TO}, l{\downarrow}(\mu') \rangle & e = \texttt{get}(\mu', f, \overline{\mu''}) \\ \langle \mathrm{TO}, l{\downarrow}(\mu') \rangle & e = \texttt{set}(\mu', f, \overline{\mu''}) \\ \langle \mathrm{TO}, l{\downarrow}(\mu') \rangle & e = \texttt{init}(A, \overline{\mu''}, \mu') \\ \bot & \textit{otherwise} \end{cases}$$

$$\nu_{\mathrm{AO}}(\texttt{entry}(i, j, m, \mu, e)) = \langle \mathrm{AO}, l{\downarrow}(\mu) \rangle$$

Figure 7. Projection (π_p), the view identification predicate ($p_{\nu_\phi|\gamma}$), and entry to view name mappings (ν_ϕ)

trace until both traces are the same length. The resulting augmented trace syntax is presented in Fig. 8, along with an extended object representation. Note that locations by themselves are unsuitable for comparison across different program versions. We thus extend object representations to tuples which now include, besides their location in the case of non-value objects, an identifier (or hash) that represents a recursively computed value representation.

$$\begin{array}{lll} \textit{object} \ \mu' & ::= & \langle l, r \rangle \\ \textit{serialization} \ r & ::= & D{:}[d] \mid C{:}[\overline{r}] \\ \textit{trace entry} \ \tau' & ::= & \texttt{entry}(eid, tid, m, \mu, e) \mid \texttt{eof} \end{array}$$

$$\mathcal{E}'{\downarrow}(D(d)) = \langle \_, D{:}[d] \rangle \qquad \frac{\mathcal{E}(l) = [f_1{:}v_1, .., f_n{:}v_n]}{\mathcal{E}'{\downarrow}(l(C)) = \langle l, C{:}[\mathcal{E}'{\downarrow}(v_1), .., \mathcal{E}'{\downarrow}(v_n)] \rangle}$$

Figure 8. Extended trace syntax and object representation used for differencing.

One important aspect to effective view-based differencing (or any analysis operating over views from multiple executions or versions) is in the formulation of effective *view correlation functions* ($\mathbb{X}$), which are used to determine if a given view in one execution trace semantically corresponds to a given view in a different execution trace. Fig. 9 defines the type signature for all correlation functions, as well as some notation and helper relations for our differencing evaluations. One correlation function needs to be defined for each view type. The correlation function accepts two trace entries instead of two view names as arguments, because the correlation function may be context-sensitive (e.g., based on value representations) as opposed to solely the view names.

Thread view correlation ($\mathbb{X}_{\mathrm{TH}}$) is determined by considering all possible thread correlations (pairs of threads) and forming a correlation with the "closest match" based on the spawning call stack of the thread (and the thread's ancestors). Method correlation ($\mathbb{X}_{\mathrm{CM}}$) correlates two methods if their full type signatures are equal. Target object and active object correlation ($\mathbb{X}_{\mathrm{TO}}$ and $\mathbb{X}_{\mathrm{AO}}$) correlate two objects if either the value representations or class-specific object creation sequence number (derivable from trace data) are equal.

Because view correlations attempt to identify relationships among program abstractions (i.e., threads, methods, objects) found across executions based *only* on the structure of their views, they are best regarded as heuristics. Nonetheless, experimental results (Section 5) show that the implemented correlation functions are effective in practice for regression cause analysis. We believe other correlation definitions may be useful for other kinds of analyses.

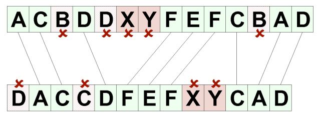


Figure 10. The LCS of two strings. Note that moved subsequences (e.g., "XY") are not detected.

$$\boxed{\langle \overline{\tau}^L, \overline{\tau}^R, \Lambda \rangle \xrightarrow[\mathcal{L}]{} \langle \overline{\tau}^{L'}, \overline{\tau}^{R'}, \Lambda' \rangle}$$

$$\frac{\Lambda' = \begin{cases} \{\tau^1\} & \tau^1 \in lcs(\overline{\tau}^{O_L}, \overline{\tau}^{O_R}) \\ \{\} & \textit{otherwise} \end{cases}}{\langle \tau^1.\overline{\tau}^{L'}, \overline{\tau}^R, \Lambda \rangle \xrightarrow[\mathcal{L}]{} \langle \overline{\tau}^{L'}, \overline{\tau}^R, \Lambda \cup \Lambda' \rangle} \ (\text{STEP-LEFT-LCS})$$

(Similar for (STEP-RIGHT-LCS) rule)

Figure 11. LCS comparison; $\xrightarrow[\mathcal{L}]{}$ is parameterized over $\overline{\tau}^{O_L}, \overline{\tau}^{O_R}$, which are defined to be the traces produced by evaluation of the two programs to be compared, before evaluated under $\xrightarrow[\mathcal{L}]{}$ or $\xrightarrow[\mathcal{V}]{}$.

We present two trace differencing semantics — one leveraging the longest common subsequence (LCS) algorithm, and the other leveraging our views trace abstraction.

3.2 LCS-based Trace Differencing Semantics

Well-known differencing tools such as Unix `diff` are founded on longest common subsequence (LCS) algorithms [3] in order to determine a minimal set of differences between two sequences (e.g., lines of text in two files). Fig. 10 visualizes how the LCS identifies the differences between two strings. Fig. 11 presents an evaluation semantics that leverages the LCS of the two traces in order to determine which trace entries should be placed in Λ. Note that two trace entries are considered to correspond based on the equality predicate, $=_e$. The LCS evaluation relation relies on preserving the original traces before any evaluation takes place. The computation of the LCS also results in a correspondence mapping between all common entries. This allows each contiguous run of differences in the traces to be viewed as either an insertion, deletion, or modification. Using the LCS to understand differences between program traces is beneficial for correlating similar yet not exactly identical events. For example, if a new version of a program adds a new parameter to a function, the LCS will gravitate towards correlating identical values, thereby identifying the new parameter as the one difference.

However, there are two major challenges in applying the LCS algorithm to execution traces. First, the algorithm for computing the LCS is not aware of program semantics and may blindly correlate neighboring entries in the original trace with entries very far apart in the new trace (e.g., consider commonly occuring values, such as 0 or `null`). Second, the computational complexity of bare LCS is $\Omega(n^2)$ [3], making it intractable on long program traces. Faster solutions are known if the input "alphabet" is fixed, but are not applicable in the present case, as the input alphabet includes value representations, which are innumerable. The standard dynamic programming algorithm requires $O(n^2)$ space in order to reconstruct the LCS (not just its length). The algorithm becomes impractical when applied to longer program traces, requiring huge memory besides time to compute the LCS. Existing algorithms with reduced space complexity require roughly twice the computation time [9].

Event equality	$\tau^1 =_e \tau^2$	True if the underlying primitive values (including those generated by $\mathcal{E}\downarrow$ in Fig. 8) of the events of the two entries are equal	
Trace index	$index(\overline{\tau}, \tau)$	The index of the entry in $\overline{\tau}$ which has an eid that matches the eid of the entry τ	$index(\overline{\tau}, \mathtt{entry}(j,k,m,\mu,e)) = \begin{cases} i & \overline{\tau} = \tau^1 \ldots \tau^{i-1}.\mathtt{entry}(j,n,m',\mu',e')\ldots \\ -\infty & otherwise \end{cases}$
Trace intersection	$\overline{\tau} \,\overline{\cap}_{=_e}\, \overline{\tau}'$	$\overline{\tau}$ with only those elements that are also in $\overline{\tau}'$ according to event equality ($=_e$)	$(\tau^1.\overline{\tau}) \,\overline{\cap}_{=_e}\, \overline{\tau}' = \begin{cases} \tau^1.(\overline{\tau} \,\overline{\cap}_{=_e}\, \overline{\tau}') & \exists \tau^i \in \overline{\tau}' : \tau^1 =_e \tau^i \\ \overline{\tau} \,\overline{\cap}_{=_e}\, \overline{\tau}' & otherwise \end{cases}$
Trace window	$win_{(\tau,\Delta)}(\overline{\tau})$	Trace $\overline{\tau}$ with only those elements whose index is in the range $[index(\overline{\tau}, \tau) \pm \Delta]$ (Δ constant, $\tau \in \overline{\tau}$)	$win_{(\tau,\Delta)}(\overline{\tau}) = \pi_{p_\Delta}(\overline{\tau})$ $p_\Delta(\mathtt{entry}(i,j,m,\mu,e)) = (i \in [index(\overline{\tau},\tau) \pm \Delta])$
Correlation	$\mathbb{X}_\phi(\tau^1, \tau^2)$	Returns $\langle \nu_\phi(\tau^1), \nu_\phi(\tau^2) \rangle$ if the views of type ϕ of the two entries are correlated or $\langle \bot, \bot \rangle$ otherwise	View correlation functions, as described in Section 3.1
LCS	$lcs(\overline{\tau}, \overline{\tau}')$	Longest common subsequence of $\overline{\tau}$ and $\overline{\tau}$' with respect to event equality ($=_e$)	Refer to [3] for details

Figure 9. Helper relations for trace differencing evaluation semantics.

$$\boxed{\langle \overline{\tau}^L, \overline{\tau}^R, \Lambda \rangle \underset{\mathcal{V}}{\longrightarrow} \langle \overline{\tau}^{L'}, \overline{\tau}^{R'}, \Lambda' \rangle}$$

$$\frac{\tau^1 =_e \tau^2}{\langle \tau^1.\overline{\tau}^{L'}, \tau^2.\overline{\tau}^{R'}, \Lambda \rangle \underset{\mathcal{V}}{\longrightarrow} \langle \overline{\tau}^{L'}, \overline{\tau}^{R'}, \Lambda \cup \{\tau^1, \tau^2\} \rangle} \quad \text{(STEP-VIEW-MATCH)}$$

$$\frac{\begin{array}{c} \Lambda' = \{\tau^r \mid LinkedSimilarEntries(\tau^1, \tau^3, \tau^r)\} \\ \tau^1 \neq_e \tau^3 \quad \tau^2 =_e \tau^4 \quad \overline{\tau}^{L'} \overline{\cap}_{=_e} \overline{\tau}^{R'} = \langle\rangle \end{array}}{\begin{array}{c} \langle \tau^1.\overline{\tau}^{L'}.\tau^2.\overline{\tau}^{L''}, \tau^3.\overline{\tau}^{R'}.\tau^4.\overline{\tau}^{R''}, \Lambda \rangle \underset{\mathcal{V}}{\longrightarrow} \\ \langle\ \tau^2.\overline{\tau}^{L''}, \tau^4.\overline{\tau}^{R''}, \Lambda \cup \Lambda' \rangle \end{array}} \quad \text{(STEP-VIEW-NOMATCH)}$$

$$\frac{\begin{array}{c} index(\overline{\tau}^{V_L}, \tau^5) \in \left[index(\overline{\tau}^{V_L}, \tau^1) \pm \Delta\right] \\ index(\overline{\tau}^{V_R}, \tau^6) \in \left[index(\overline{\tau}^{V_R}, \tau^3) \pm \Delta\right] \\ \langle \gamma^L, \gamma^R \rangle = \mathbb{X}_\phi(\tau^5, \tau^6) \\ \tau^r \in lcs(win_{(\tau^5,\Delta)}(\pi_{p_{\nu_\phi \mid \gamma^L}}(\overline{\tau}^{O_L})), \\ \qquad win_{(\tau^6,\Delta)}(\pi_{p_{\nu_\phi \mid \gamma^R}}(\overline{\tau}^{O_R}))) \end{array}}{LinkedSimilarEntries(\tau^1, \tau^3, \tau^r)} \quad \text{(SIMILAR-FROM-LINKED-VIEWS}_\phi\text{)}$$

Figure 12. View-based comparison; $\underset{\mathcal{V}}{\longrightarrow}$ is parameterized over $\overline{\tau}^{O_L}, \overline{\tau}^{O_R}, \overline{\tau}^{V_L}, \overline{\tau}^{V_R}$. We define $\overline{\tau}^{V_L}, \overline{\tau}^{V_R}$ to be the two thread views being compared, but before being evaluated under $\underset{\mathcal{V}}{\longrightarrow}$.

3.3 Views-based Trace Differencing Semantics

The aforementioned challenges of LCS can be overcome by leveraging our views trace abstraction. Instead of differencing the raw pair of traces produced by the contextual semantics, we instead apply a trace differencing evaluation semantics to one or more pairs of correlated thread views. Evaluation of each pair of views ($\underset{\mathcal{V}}{\longrightarrow}$) proceeds by removing the head elements and if they are equal, placing them in set Λ. For the other case where the head elements are different, any secondary views linked to nearby entries in the main views are explored to find correlations between entries in the left and right traces, and then removing from the heads of the left and right main views any entries up until the next common point of correlation.

Fig. 12 formalizes the above intuition of how one pair of thread views is evaluated. The rule (STEP-VIEW-MATCH) handles the case where the heads of the evaluated thread view pair have equal events, in which case the entries are removed and added to Λ. The (STEP-VIEW-NOMATCH) rule handles the other case. It adds to Λ those entries that were found to be similar by comparing corresponding views nearby the differing entries, as modeled by $LinkedSimilarEntries$. The evaluation step also removes from the heads of the traces any differing entries up until the next pair of similar entries. Evaluation thus alternates between the two rules until the end of the traces is reached.

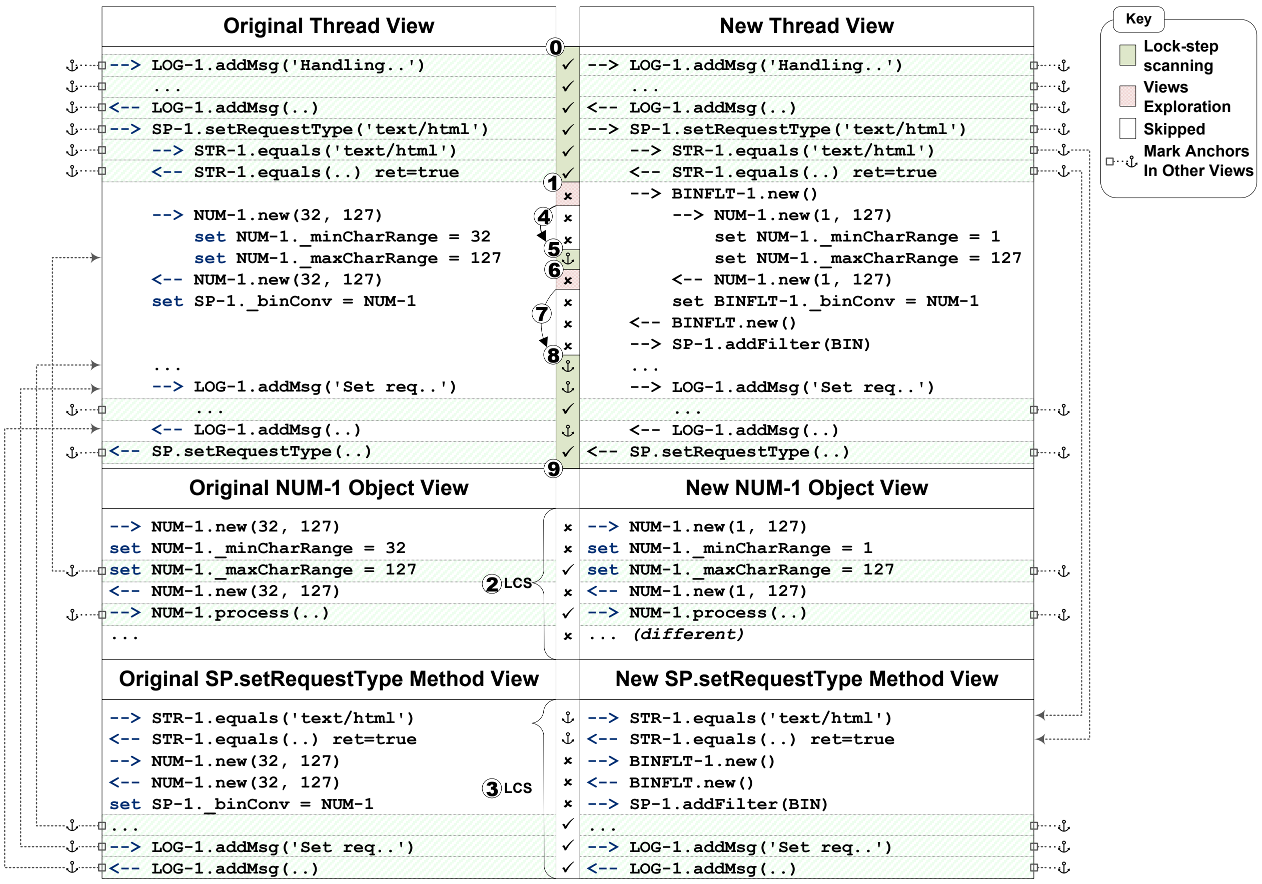
The generic rule (SIMILAR-FROM-LINKED-VIEWS$_\phi$) defines how to construct $LinkedSimilarEntries$ for a given view type, ϕ. Not shown are the concrete rules, one for each view type (with ϕ instantiated to TH, CM, TO, or AO). To further explain the $LinkedSimilarEntries$ relation, for a given pair of trace entries from the left and right thread views (τ^1,τ^3) for whom secondary views should be explored to identify similar entries, a similar entry τ^r is identified according to the antecedent of (SIMILAR-FROM-LINKED-VIEWS). The first and second lines of the antecedent constrain the free variables τ^5 and τ^6 to be trace entries within a constant distance from the entries τ^1 and τ^3 respectively. The third line requires that entries τ^5 and τ^6 either have corresponding thread views, method views, or object views (which views the two entries have in correspondence are called matching views). Line four then constrains free variable τ^r such that it must be in the LCS of fixed-size windows of the matching views. In this way, similar entries in corresponding secondary views are identified using LCS, but over fixed-size windows of the nearby entries in the views as opposed to the entire view or the entirety of the raw traces.

When differencing a raw pair of traces produced by a program, we evaluate each pair of corresponding thread views (as determined by $\mathbb{X}_{\text{TH}}$) under $\underset{\mathcal{V}}{\longrightarrow}$, each producing a different set Λ. These sets are unioned together to form the final similarity set, Λ_f. The final set of differences is then derived from Λ_f by set subtraction. We have shown our technique to exhibit $O(n)$ complexity in both space and time; the proof has been omitted due to space restrictions and is available in a tech report [10].

3.4 Example

Fig. 13 shows how the differencing algorithm works for the motivating example shown in Fig. 1. As each trace evaluation rule removes one or more entries from the head of the traces, the numbered circles show the progression of the evaluation as rules are applied. From the start of the trace up to and including the call to the `setRequestType` method (from circle 0 to 1), the (STEP-VIEW-MATCH) is applicable, adding these entries to set Λ. At circle 1, the (STEP-VIEW-NOMATCH) rule becomes applicable. τ^1 and τ^3 correspond to entry 9 in the left and right traces, respectively (the fact that the entry id is the same is coincidental here).

$LinkedSimilarEntries$ for (τ^1,τ^3) identifies those entries that should be placed in set Λ due to the exploration of corresponding secondary views nearby to these entries. The two secondary views displayed in the figure represent two such corresponding views. Circles 2 and 3 depict the relevant fixed windows in the secondary views over which the LCS is computed, and the checkmarks in


Original Thread View
New Thread View
Key
Lock-step scanning
Views Exploration
Skipped
Mark Anchors In Other Views
Original NUM-1 Object View
New NUM-1 Object View
LCS
Original SP.setRequestType Method View
New SP.setRequestType Method View
LCS
0
1
4
5
6
7
8
9
2
3

```
Original Thread View                          | New Thread View
--> LOG-1.addMsg('Handling..')                | --> LOG-1.addMsg('Handling..')
    ...                                       |     ...
<-- LOG-1.addMsg(..)                          | <-- LOG-1.addMsg(..)
--> SP-1.setRequestType('text/html')          | --> SP-1.setRequestType('text/html')
    --> STR-1.equals('text/html')             |     --> STR-1.equals('text/html')
    <-- STR-1.equals(..) ret=true             |     <-- STR-1.equals(..) ret=true
                                              |     --> BINFLT-1.new()
    --> NUM-1.new(32, 127)                    |         --> NUM-1.new(1, 127)
        set NUM-1._minCharRange = 32          |             set NUM-1._minCharRange = 1
        set NUM-1._maxCharRange = 127         |             set NUM-1._maxCharRange = 127
    <-- NUM-1.new(32, 127)                    |         <-- NUM-1.new(1, 127)
    set SP-1._binConv = NUM-1                 |         set BINFLT-1._binConv = NUM-1
                                              |     <-- BINFLT.new()
                                              |     --> SP-1.addFilter(BIN)
    ...                                       |     ...
    --> LOG-1.addMsg('Set req..')             |     --> LOG-1.addMsg('Set req..')
        ...                                   |         ...
    <-- LOG-1.addMsg(..)                      |     <-- LOG-1.addMsg(..)
<-- SP.setRequestType(..)                     | <-- SP.setRequestType(..)
```

```
Original NUM-1 Object View                    | New NUM-1 Object View
--> NUM-1.new(32, 127)                        | --> NUM-1.new(1, 127)
set NUM-1._minCharRange = 32                  | set NUM-1._minCharRange = 1
set NUM-1._maxCharRange = 127                 | set NUM-1._maxCharRange = 127
<-- NUM-1.new(32, 127)                        | <-- NUM-1.new(1, 127)
--> NUM-1.process(..)                         | --> NUM-1.process(..)
...                                           | ... (different)
```

```
Original SP.setRequestType Method View        | New SP.setRequestType Method View
--> STR-1.equals('text/html')                 | --> STR-1.equals('text/html')
<-- STR-1.equals(..) ret=true                 | <-- STR-1.equals(..) ret=true
--> NUM-1.new(32, 127)                        | --> BINFLT-1.new()
<-- NUM-1.new(32, 127)                        | <-- BINFLT.new()
set SP-1._binConv = NUM-1                     | --> SP-1.addFilter(BIN)
...                                           | ...
--> LOG-1.addMsg('Set req..')                 | --> LOG-1.addMsg('Set req..')
<-- LOG-1.addMsg(..)                          | <-- LOG-1.addMsg(..)
```

Figure 13. How the views-based trace differencing semantics works for part of our example. The primary view and two secondary views are shown. Evaluation progression is labeled with circles 0–9. ✘ denotes entries that differ. ✓ denotes entries placed in set Λ via evaluation of the (Step-View-Match) rule. ⚓ denotes entries placed in set Λ via evaluation of the (Step-View-NoMatch) rule. For example, the thread view entry at circle 5 (`set NUM-1._maxCharRange = 127`) was marked with ⚓ due to comparisons within the `NUM-1` object view.

these views depict which entries are thereby identified as corresponding (and thus should be in set Λ). Entries thus identified as corresponding due to exploration in secondary views are denoted with the anchor symbol in the figure.

Note that even though these entries appear "nearby" to the current positions in the thread views in this example, in large traces these entries identified as similar from secondary views could be thousands or hundreds of thousands of trace entries away. In this way exploration of secondary views allows for recognizing semantically correlating events that could be very far apart in the thread views. This characteristic is one reason this approach allows for greater accuracy than LCS, as this approach remains resilient to reorderings of operations in the thread views whereas LCS identifies these reorderings as differences.

With $LinkedSimilarEntries$ fully formed for the given point (τ^1,τ^3), the antecedent of (Step-View-NoMatch) identifies the next closest point of correspondence (circle 5) in the left and right thread views (τ^2,τ^4) and removes all entries from the heads of the traces up to but not including this next point of correspondence (circle 4).

Evaluation thus proceeds alternating between these two rules until the end is reached: at circle 5, (Step-View-Match) is applied, placing entry 11 (left) and entry 12 (right) into set Λ and proceeding to circle 6. Here (Step-View-NoMatch) is applied again, $LinkedSimilarEntries$ is formed again for this unique point (thereby exploring nearby secondary views that correspond), and identifying the next point of correspondence as the position at circle 8. The (Step-View-Match) is then applied for all remaining traces until the end is reached at circle 9.

4. Regression Cause Analysis

We envision many types of dynamic analyses benefiting from our views trace abstraction and our views-based trace differencing semantics, including object protocol inference, property checking (e.g., typestate), impact analysis, and automated debugging. As mentioned we focus our attention in this paper on how semantics-aware trace differencing empowers regression-cause analysis.

4.1 Algorithm

To identify the causes of regressions, we employ our semantics-aware trace differencing to identify all the semantic differences between an original, non-regressing version and a new, regressing version of a program for one or more regressing test cases. Let A represent this set of semantic differences, termed the *suspected differences set*. The size of set A is usually too large to be effective for finding the regression cause through manual inspection (having in practice hundreds or thousands of differences). The goal of the analysis algorithm is thus to remove from set A those differences that are *not* likely to have caused the regressing behavior. The algorithm proceeds as follows:

1. A set of differences that are expected to occur under correct execution between the original and the new version is built (call this set B), by comparing execution traces for runs of the two versions for test case(s) with *correct* behavior. These differences are not likely to be related to the cause of the regression, because the regressing behavior was not observed during execution of these correct test case(s). This set B is the *expected differences set*.
2. A set of differences for the new version between a correct test case and the regressing test case is built (C). This set of differences includes the difference(s) that cause the regressing behavior. The correct test case should be similar in functionality to the regressing test case so that the resulting set of differences is small and yet still includes the difference(s) causing the regression (this is similar to the requirements of [21] and can be computed automatically using techniques from [22]). This set C is termed the *regression differences set*.
3. The set of differences between the original and new version that are highly likely to be responsible for the cause of the regression, termed D, is now calculated as follows:

$$D = (A - B) \cap C$$

The choice of set B impacts the effectiveness of this approach because the cause for a regression can appear within the execution trace for non-regressing test cases. Eliminating the differences may thereby eliminate the cause, introducing false negatives.

In practice, however, we find that filtering differences as described does not compromise accuracy for three reasons. First, our dynamic traces are complete, so the cause of the regression must be in A, implying there are no false negatives at this stage. Second, eliminating the expected differences (B) eliminates many false positives, increasing accuracy. Since these differences did not exhibit within the program output, it is highly likely they are related to correct program evolution instead. Third, intersecting with the regression differences set (C) further eliminates false positives. As C only contains differences within the new program version, there are fewer differences (only those caused by the differing inputs of the regressing and non-regressing test cases and excludes any differences due to program evolution), and yet is still likely to contain the regression cause.

The other source of false negatives is due to $A \cap C$. If the regression is caused by the removal of code in the new version, then there is no possibility for set C to contain the regression-inducing differences. For these cases, the analysis can be changed to:

$$D = (A - B) - C$$

Subtracting set C for these cases will further reduce false positives without introducing false negatives, allowing the analysis to effectively find regression causes due to the removal of code in newer versions. We empirically evaluate the effectiveness of our approach in practice in Section 5.

4.2 Example Revisited

Recall that a regression is caused in our example when the class `BinaryCharFilter` is instantiated with the incorrect range of [1..127] instead of [32..127]. The most interesting code fragments related to this regression are depicted in Fig. 1.

Even though there are many differences in the new version of the code, only seven are relevant to the regression and its cause. Our tool correctly identifies these seven changes, with no false positives. It also correctly identifies 20 other runs of differences as not being related to the regression.

To achieve these results we created two test cases: (a) a test case that reproduced the regression, and (b) a test that used a different document type, so conversion of the characters was not applied in both versions (and thus does not exhibit the regressing behavior). First, we collected dynamic traces for the original and new versions for both test cases. Next, the semantic differencing tool was run on the following pairs of traces: (i) original version vs. new version for the regressing test case (forms the *suspected differences set*); (ii) original version vs. new version for the non-regressing test case (forms the *expected differences set*); (iii) a new version of the non-regressing test case vs. a new version of the regressing test case (this forms the *regression differences set*).

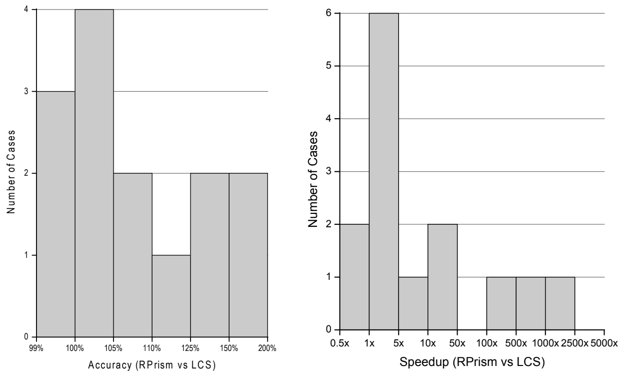
5. Evaluation

RPRISM employs aspect-oriented programming (AOP) [12] to dynamically instrument Java programs via AspectJ's load-time weaver. *Pointcuts* provide a flexible mechanism to capture the segments of the execution trace that should be recorded. The implementation is layered and provides the following features: trace segmentation, call stack tracking, event recording, view construction, trace serialization, and tracing control. Tracing of long-running programs are accomodated through smart trace segmentation–AspectJ pointcuts are used to specify relatively short regions of program execution to record as a single trace segment, and once a trace segment has finished executing, all trace data is offloaded to disk and the associated tracing memory is reclaimed. While the program is running the execution trace is collected but not analyzed – the analysis is performed offline after the trace data has been serialized to disk.

RPRISM implements the view correlation of Section 3.1, and also relaxes method and object view correlation (to be more tolerant to refactorings) using a context-sensitive correlation function that correlates views if their entries are the same "distance" (number of trace entries) away from two points in the traces that are known to be semantically correlated to each other. This relaxation improves robustness in the presence of refactorings, such as if methods or classes have been renamed, or methods have been split or combined. For example, if a method has been renamed, then there will be a difference at the point of the call site, but it is probable that there will be call sites in the versions where either the immediately preceding or succeeding statement is semantically correlated. Under the relaxation, the secondary views will be explored at the point of the call site of the renamed method. When the secondary views are analyzed, the (mostly) unchanged code in the method whose name was refactored produces many new semantic correlations back in the main view, thus providing tolerance to the method rename refactoring.

RPRISM approximates the value representations in our formalism using the Java `hashCode` and `toString` (truncated to 128 chars) methods, which we found is a good approximation in practice, providing both reasonable performance and accuracy. Note that if an object uses the default `java.lang.Object` implementation of these methods, the value representation is forced to be empty as it is not meaningful across different program versions.

In this section we first quantitatively assess the effectiveness of our semantics-aware trace differencing, comparing it to an optimized version of LCS with respect to both accuracy and performance. The differing software versions for this assessment are based on the iBUGS project [5]. We further evaluate our regression-cause analysis by discussing in detail RPRISM's accuracy and performance on four real-life regressions. All tests were executed on a server with eight 1.8Ghz dual-core Opteron processors and 32GB of RAM. Note that RPRISM is currently single-threaded, with the extra cores only in use during the VM's parallel garbage collection.



(a) Accuracy (b) Speedup

Figure 14. Quantitative results based on iBugs Rhino dataset comparing RPRISM with an optimized LCS implementation.

5.1 Quantitative Assessment

Our goal is to assess both (i) the semantics-aware trace differencing, and (ii) the regression-cause analysis. Existing research bug databases contain few if any real regressions (focusing on explicit bugs instead). Without any immediately suitable benchmark regression bug database, we built on the Rhino dataset in the iBUGS project [5]. The Rhino dataset is a set of 29 bugs from the Mozilla Rhino project,[6] which implements JavaScript in Java and consists of 304 KLOC (including tests), 242 classes, and 15K tests. Rhino compiles JavaScript into an intermediate form, which is then either interpreted or compiled to standard Java classes. Our data here is based on the interpretive mode, as it produced longer and more complex traces, but RPRISM runs equally well with the compiled mode.

We integrated RPRISM into the automated build process, providing unattended runs over all bugs. We introduced regressions into each post-fix version by either using the actual cause of the bug itself if the bug was a regression or by using a distribution of root causes that matches the distribution found for semantic bugs in the Mozilla project in an empirical study [13]. Root causes considered are categorized as missing features (26.4%), missing cases (17.3%), boundary conditions (10.3%), control flow (16.0%), wrong expressions (5.8%), or typos (24.2%). We ensured that each injected regression caused the test case associated with the bug to fail.

We utilized RPRISM to trace the working and regressing versions and to calculate the differences between the traces. As we are modeling what RPRISM would provide to developers if it were part of a completely automated build process, we do not follow the final step of manually creating similar non-regressing test cases, which would only increase accuracy further. A developer could complete this final step if they wished to further refine RPRISM output when fixing the regression.

[6] `http://www.mozilla.org/rhino/`

Measurements. To assess the effectiveness of our view-based trace differencing technique we define two measures: accuracy and speedup. Accuracy measures how many semantic correlations are identified with RPRISM vs the number of correlations identified with the LCS comparison. Accuracy is precisely defined using the following formula:

$$\text{Accuracy} = \frac{((\text{totalEntries} - \text{rprismNumDiffs})/\text{totalEntries})}{((\text{totalEntries} - \text{lcsNumDiffs})/\text{totalEntries})}$$

Note that because RPRISM can identify reorderings of operations it often identifies fewer semantic differences than LCS (resulting in more semantic correlations), resulting in an accuracy value greater than 100%. An accuracy of 100% in this section should be read as meaning "RPRISM identified the same number of semantic differences as in the LCS comparison." Speedup is defined as the number of trace entry compare operations performed during the LCS comparison divided by the number of compare operations performed during comparison with RPRISM.

Results. Salient measurements from our experiments are shown in Fig. 14. Most traces were between 10K and 100K entries, with a few outliers ranging up to 1.9 million entries. Trace size was optimized by leveraging AspectJ pointcuts to exclude the internal workings of unrelated code, such as libraries and data structures.

RPRISM organizes contiguous sets of differences into *difference sequences*, thereby organizing tool output into comprehensible units. More than two-thirds of the bugs produced less than 50 difference sequences, with the remainder ranging from 50 to 130 difference sequences. We found that the cause of the divergence is often observed in the first handful of differences sequences. If further refinements are required, an alternate but non-regressing test case can be created, and then the number of difference sequences is typically cut by an order of magnitude or more (see next section).

We calculated the precise LCS where possible using an optimized version of the LCS algorithm (common-prefix/suffix optimizations). Fig. 14(a) measures accuracy by comparing the relative number of trace entries marked as different by each approach. RPRISM achieves greater than 100% accuracy in all but 3 cases because it is able to make semantically correct correlations (such as detecting moved trace entries) that the LCS is inherently not able to detect. The remaining 3 cases had accuracy greater than 99%.

We evaluate RPRISM's efficiency by measuring speedup relative to the number of compare operations (Fig. 14(b)). The LCS approach failed on traces longer than 100K entries (due to memory exhaustion), whereas RPRISM successfully analyzed traces as long as 1.9 million entries. RPRISM achieved speedups of more than 100x vs the LCS algorithm. For two very small traces RPrism had speedups less than 1x, because of the extra comparisons in secondary views. Observed wall clock times for trace differencing are also reasonable – running time of the algorithm took less than 1 second in all cases, except for the trace with 1.9 million entries, which took 110 seconds.

Note that the histograms include data for only 14 of the 29 bugs in the iBugs Rhino suite, for the following reasons: Two bugs were not runnable due to a problem with the iBugs distribution. The LCS algorithm failed due to memory exhaustion for 3 bugs. The remaining bugs had trouble generating tracing data, due to problems with the AspectJ weaver (invalid bytecode was produced by the AspectJ weaver).

5.2 Real-life Regressions

To assess RPRISM's utility in identifying regression causes, we analyze four previously documented regressions in significantly

				LCS-based Differencing							Views-based Differencing							
Benchmark	LOC	Trace Entries	Tracing Secs.	Num Diffs.	Diff. Seqs.	Regression Diff. Seqs.	False Pos.	False Neg.	Analysis Secs.	Mem (GB)	Num Diffs.	Diff. Seqs.	Regression Diff. Seqs.	False Pos.	False Neg.	Analysis Secs.	Mem (GB)	Speedup
Daikon	169K	15K	185	352	43	3	0	1	44	0.85	179	42	3	0	1	3.4	0.07	12.9x
Xalan-1725	365K	98K	90	2,338	145	0	0	1	1,515	26.9	1,197	296	1	0	0	18.3	0.10	82.8x
Xalan-1802	286K	44K	99	4,269	117	11	0	0	582	7.43	3,602	184	10	0	0	61.5	0.21	9.4x
Derby-1633	720K	337K	182	(out of memory failure at 32GB)							125,562	2,663	6	4	0	80.1	0.34	-

Table 1. Benchmark and analysis characteristics (time/memory are median of 3 runs).

	Number of Views				Size of Analysis Sets			
Benchmark	Total views	Thread views	Method views	Target object views	A	B	C	D
Daikon	559	1	127	431	42	NA	22	3
Xalan-1725	1,679	1	446	1,232	296	243	113	1
Xalan-1802	1,811	1	497	1,313	184	183	10	10
Derby-1633	6,874	3	1,761	5,110	2,663	4	310	10

Table 2. Number of views (in the original program version only) and size of the sets in the regression-cause analysis algorithm for our benchmarks. Set A is the suspected differences set, set B is the expected differences set, set C is the regression differences set, and set D is the result of the analysis.

sized open-source software projects, namely Daikon [6], Apache Xalan[7] (2 regressions), and Apache Derby.[8] Our reasons for choosing these regressions are as follows: The Daikon regression was chosen because this exact same regression was also evaluated by JUnit/CIA [17], providing a comparison to a previously established method for regression-cause analysis. The first Xalan regression was chosen because it involved an extreme separation of cause and effect, as the cause is within a dynamic bytecode compiler and the visible effects are not exhibited until the compiled bytecode is executed. The second Xalan regression was chosen because the cause of the regression was in a completely rearchitected module in the code, and exhibited a large amount of code churn in general (79K lines); we wanted to observe how this level of code churn would affect accuracy. The Derby regression was chosen because it involved multiple threads, a larger code base (2x), and offered larger, longer-running traces (3x) than the other regressions.

The versions to use when analyzing each regression were determined as follows: With Daikon, we used the versions as published in the evaluation of JUnit/CIA. For the others, we chose the last publicly released version that was working correctly, and the first publicly released version after the regression was reported but not yet fixed (at least 5 months between the versions chosen in all cases).

Table 1 summarizes characteristics of the benchmarks and our analysis results. For comparison, we present the results of the regression analysis based on both the LCS-based and view-based differencing semantics. *Num Diffs.* states the number of distinct differences identified. *Diff. Seqs.* states how many difference sequences (each representing one higher-level semantic difference that manifests as a contiguous set of differences) were formed from the raw differences. *Regression Diff. Seqs.* states how many of these sequences were identified by RPRISM as being regression related. *False Pos.* states the number of semantic differences incorrectly identified as regression-related by RPRISM; *False Neg.* states the number of regression-inducing differences (as identified by the developers in the bug description) that exist between the non-regressing and regressing versions that were not identified. Speedup is based on wall clock time of the differencing analysis. Table 2 summarizes the number of views and the sizes of the regression analysis sets. Note that the contents of sets A, B, C, and D can all be very different, which is why $|D|$ can be much smaller than $|C|$ and why $|D|$ can be larger than $|A| - |B|$.

[7] `http://xml.apache.org/xalan-j/`

[8] `http://db.apache.org/derby/`

Daikon. Daikon is an extensible tool for dynamically detecting likely program invariants (169 KLOC, 1100 classes). We revisit a regression first considered in the evaluation for JUnit/CIA [17]. This regression is exhibited by an outdated version of the `testXor` test case. The regression was caused by changes in two methods in class `daikon.diff.XorVisitor`, namely `shouldAddInv1` and `shouldAddInv2` [17]. The older `testXor` version was selected as the regressing test case and the newer `testXor` version was selected as the non-regressing test case. Note that Daikon took the longest to trace even though it produced the shortest traces because the test case involved the most number of distinct classes, resulting in 98% of the time for tracing spent performing aspect weaving.

Out of 42 difference sequences, RPRISM identified 3 of these as relating to the regression; 2 of these differences correctly identified changes in the `shouldAddInv2` method as the regression cause, although the change in `shouldAddInv1` was not identified (a false negative). The other difference was related to the effect of the regression but not the causes. Notably, RPRISM was more precise than JUnit/CIA for this regression. Whereas JUnit/CIA also correctly identified the changed methods that caused the regression, it also labeled 2 other changes as "Red" (highly likely to be the cause), and 31 other changes as "Yellow" (changes that might be a cause) [17].

Apache Xalan. Xalan is an implementation of XSLT, an XML transformation language (365 KLOC, 1500 classes). We consider a bug from Xalan's bug database, XALANJ-1725,[9] involving a regression from version 2.5.1 to 2.5.2. Version 2.5.2 incorporates 4 months of code changes, including 84 feature enhancements and bug fixes. The cause of the regression is quite difficult to pinpoint because the bug is in the XSLT compiler (which generates Java bytecode). This is an extreme case of separation of cause and effect, as the former lies in incorrectly generated bytecode, so the latter only manifests upon execution of that bytecode. This makes it extremely difficult for any static analysis technique to accurately understand and identify precisely both cause and effect.

The bug report provided an XSLT file that was correct on version 2.5.1 but not on version 2.5.2. To obtain a similar non-regressing test case, we modified the XSLT file and removed the small section of the file that was causing incorrect behavior while leaving the remainder of the file the same, and it was constructed without foreknowledge of the regression cause. Alternatively, automated techniques could be applied to construct the alternate test case [22].

[9] `https://issues.apache.org/jira/browse/XALANJ-1725`

Views-based differencing was more precise than LCS, as it only produced about half as many differences as LCS produced. Note that the number of difference sequences is larger for views-based differencing because there are more similarities interspersed among the differences. Consequently, the views-based difference sequences tended to be shorter and more concise, as evidenced by the average number of differences per sequence (4.04 vs 16.12 for LCS). This trend is also true for all other bugs where LCS was computable, with RPRISM producing finer-grained and thereby more precise results. This finer granularity allowed the views-based differencing to precisely identify the regression cause of this bug, whereas the LCS-based differencing failed to produce any regression differences.

RPRISM identified 296 semantic differences between the original and new versions for the regressing test case. After applying our regression cause analysis algorithm the suspected differences set was reduced to 1 difference. This identified difference is in the `checkAttributesUnique` method and called by the `LiteralElement.translate` method, which was identified as the regression cause in the Xalan bug database.

Xalan-1802. We consider another regression in Xalan exhibited between versions 2.4.1 and 2.5.1 (12 months, 79K new or changed lines of code, 97 bugfixes/feature changes).[10] As before we generated a similar, non-regressing input file from the regressing input file provided with the bug. In this case the regression was caused not by small incremental changes but by a bug for a corner case in a completely re-architected portion of the code relating to namespaces. Note that the views-based analysis took more time for this bug than the other Xalan bug even though the traces were half the size, because there were more differences in the main views, and consequently more secondary views had to be explored during analysis. Note that this trend does not apply to the LCS-based approach, whose running time is approximately quadratic with respect to the trace size (sometimes less because of optimizations).

Apache Derby. Derby is a mature Java implementation of a relational database system. We consider a regression from version 10.1.2.1 to 10.1.3.1 relating to query predicates and subqueries, as documented in Derby bug DERBY-1633.[11] The bug description provided a regressing SQL query and sample database. We formed the alternate test case by modifying the predicate causing the regression in the SQL query. Both code size and trace size here are substantially larger than for the other regressions we considered. Of unique note here is that Derby is multithreaded and generated multiple thread views during tracing. Our views trace abstraction allowed for proper analysis and elimination of behavior on other threads not related to the regression. The bug was caused by an incomplete corner case in new query optimizations introduced in version 10.1.3.1. The large number of differences (125K) was caused by observing version 10.1.2.1 executing the query vs 10.1.3.1 throwing an error during query compilation. The analysis algorithm was able to effectively eliminate these regression side-effects and non-relevant differences, instead identifying 6 difference sequences all directly related to the regression (as confirmed by reviewing the posted code patch for the regression). Four false positives were also observed, relating to differences from use of database locks that were not related to the regression cause.

[10] `https://issues.apache.org/jira/browse/XALANJ-1802`

[11] `https://issues.apache.org/jira/browse/DERBY-1633`

5.3 Impact of Code Refactorings on Accuracy of RPRISM

Refactorings and greater chronological distance between the working and regressing version certainly increase the size of the suspected differences set (set A). The differences due solely to refactoring or other modifications that are not regression-related are modeled by the expected differences set (set B) (difference between non-regressing test cases for the two versions). When set B is subtracted from set A, this removes most of these unrelated differences. Furthermore, intersection with the regression differences set (between similar regressing and non-regressing tests on the *same* version) also serve to further eliminate other unrelated refactoring changes. Our evaluation shows that even in cases where there were months of active development and lots of code churn RPRISM effectively identified the regression cause with precision. For example, Xalan-1802 exhibited 12 months of active development and 79K new or changed lines of code between versions, and RPRISM correctly trimmed the suspected differences set down from 184 difference sequences to 10 difference sequences.

6. Related Work

Dynamic program slicing [1] is a technique that identifies all statements that directly or indirectly affect a variable's value for given program inputs. This produces far more information than the handful of differences produced by RPRISM. In dynamic slicing the number of statements is often measured as a percentage of executed statements, with percentages in the 0.1% to 1% range being considered excellent (e.g., see [19]). By this measure, the results for RPRISM are 0.02% (Daikon), 0.001% (Xalan-1725), 0.0035% (Xalan-1802), and 0.003% (Derby-1633).

Execution indexing [20] is a technique that can be used to correlate related program points between executions. It leverages properties of an execution's dynamic state based on its nesting structure (loops, calls, etc.) to uniquely label program points. Our view-based projections can be regarded as a form of execution indexing that correlates events across different executions based on causal semantic properties (e.g., order of method calls, object allocations, object field reads and writes, etc.). Rather than using full context information to determine an index, we use anchor points derived from a derivative of an LCS calculation on program traces; our technique is especially well-suited for regression analysis over program versions. In general, there has been substantial work on software fault localization using dynamic execution traces that are also related to our work. These approaches employ a variety of techniques including statistical machine learning [14], program slicing [24, 18] control-flow similarity metrics [8], or state-space exploration and refinement [21]. These techniques cannot be easily adapted to identify regression failures between different versions of a program.

Pothier *et al.* [15] present a portable Trace-Oriented Debugger for Java which uses efficient instrumentation techniques for event generation and a scalable storage system for completeness and efficient querying. RPRISM is a complementary infrastructure for regression cause analysis. While RPRISM could query Trace-Oriented-Debugger to effectively construct views, we present an approach for implementing tracing using AspectJ, which automatically provides semantic information by identifying trace boundaries.

JDiff [2] is a tool for identifying the differences between two Java programs. In their approach, the authors use method level representation to model object-oriented features, build a representation for pairs of matching methods, and subsequently identify the differences/similarities across methods. RPRISM operates at a finer granularity to detect such differences, and is robust even in the presence of concurrency, reflection, dynamic class loading and other advanced language features.

CHIANTI is a tool for change impact analysis of Java programs by Ren *.et al* [16]. It identifies a set of changes responsible for a modified test's behavior and the set of tests that are affected by a modification. The differences between two versions are decomposed into a set of atomic changes and, based on static or dynamic call graph sequences, the above mentioned details are estimated. This system is extended in JUNIT/CIA [17] to classify which atomic changes are likely to have caused specific test cases to fail. Unlike RPRISM, this approach requires source code and is not well-suited for cases involving dynamic code generation (such as in our Xalan case study).

DSD-Crasher [4] is a proactive bug-finding technique that automatically finds bugs via a three-phase approach: (i) automatically computing program invariants via dynamic analysis, (ii) statically analyzing the code to look for possible execution paths where invariants fail, (iii) validating these potential failures through automatic test case generation.

7. Conclusion

This paper presents a novel *view-based* technique for tractably comparing large execution traces in semantically meaningful ways. We have illustrated the usefulness of this technique through the problem of precisely identifying deeply buried causes of regressions introduced in evolving complex applications.

We have presented an *automated* regression cause analysis algorithm based on our technique. RPRISM, an implementation of these ideas, was applied to large, real-world Java applications, and was able to identify the cause of regressions with a high degree of precision, even when the applications employ multi-threading, dynamic code generation, and reflection, features that confound previously proposed analyses.

Acknowledgments

We would like to thank Murali Krishna Ramanathan and the anonymous reviewers for their time and effort spent improving this paper.

References

[1] H. Agrawal and J.R. Horgan. Dynamic Program Slicing. In *PLDI'90*, pages 246–256, 1990.

[2] T. Apiwattanapong, A. Orso, and M.J. Harrold. JDiff: A Differencing Technique and Tool for Object–Oriented Programs. *ASE'07*, 14(1):3–36, 2007.

[3] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *SPIRE'00*, page 39, 2000.

[4] C. Csallner and Y. Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *ISSTA'06*, pages 245–254, 2006.

[5] Valentin Dallmeier and Thomas Zimmermann. Extraction of Bug Localization Benchmarks from History. In *ASE*, pages 433–436, 2007.

[6] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *TSE*, 27(2):1–25, 2001.

[7] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[8] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately Choosing Execution Runs for Software Fault Localization. In *CC*, pages 80–95, 2006.

[9] D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *CACM*, 18(6):341–343, 1975.

[10] Kevin Hoffman, Patrick Eugster, and Suresh Jagannathan. RPrism: Efficient Regression Analysis Using View-Based Trace Differencing. Technical Report dynt-200811-1, Purdue University, 2008.

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, pages 220–242, 1997.

[13] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *ASID*, pages 25–33, 2006.

[14] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *PLDI '03*, pages 141–154, 2003.

[15] G. Pothier, E. Tanter, and J. Piquer. Scalable Omniscient Debugging. In *OOPSLA'07*, pages 535–552, 2007.

[16] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA'04*, pages 432–448, 2004.

[17] M. Stoerzer, B. Ryder, X. Ren, and F. Tip. Finding Failure-Inducing Changes in Java Programs Using Change Classification. In *ESEC-FSE-14*, pages 57–68, 2006.

[18] F. Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3:121–189, 1995.

[19] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, 1995.

[20] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient Program Execution Indexing. In *PLDI '08*, pages 238–248, 2008.

[21] Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs. In *FSE-10*, pages 1–10, 2002.

[22] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *TSE*, 28(2):183–200, 2002.

[23] X. Zhang and R. Gupta. Cost Effective Dynamic Program Slicing. In *PLDI'04*, pages 94–106, 2004.

[24] X. Zhang and R. Gupta. Matching Execution Histories of Program Versions. In *ESEC/FSE-13*, pages 197–206, 2005.