

4-15-2006

Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software

Alex Kinneer

University of Nebraska - Lincoln, akinneer@cse.unl.edu

Matthew B. Dwyer

University of Nebraska-Lincoln, dwyer@cse.unl.edu

Gregg Rothermel

University of Nebraska - Lincoln, grothermel2@unl.edu

Follow this and additional works at: <http://digitalcommons.unl.edu/csetechreports>



Part of the [Computer Sciences Commons](#)

Kinneer, Alex; Dwyer, Matthew B.; and Rothermel, Gregg, "Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software" (2006). *CSE Technical reports*. 20.

<http://digitalcommons.unl.edu/csetechreports/20>

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

*Technical Report TR-UNL-CSE-2006-0006, Department of Computer Science and Engineering, University of Nebraska–Lincoln,
Lincoln, Nebraska, USA, 15 April 2006*

Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software

Alex Kinneer, Matthew B. Dwyer, and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, Nebraska 68588-0115, USA
{akinneer, dwyer, grother}@cse.unl.edu

15 April 2006

Abstract

Dynamic analysis techniques are well established in the software engineering community as methods for validating, understanding, maintaining, and improving programs. Generally, this class of techniques requires developers to instrument programs to generate events that capture, or *observe*, relevant features of program execution. Streams of these events are then processed to achieve the goals of the dynamic analysis. The lack of high-level tools for defining program observations, automating their mapping to efficient low-level implementations, and supporting the flexible combination of different event-stream-based processing components hampers the development and evaluation of new dynamic analysis techniques. For example, mapping non-trivial program observations to existing low-level instrumentation facilities is a time-consuming and error-prone process that can easily result in poorly performing analyses.

In this paper, we present *Sofya* - a framework that we have developed for building dynamic analysis tools. We describe the architecture of *Sofya*, and explain how it meets the challenges faced by developers of a wide-range of dynamic analyses. We survey existing dynamic analysis tools to show how they relate to the capabilities of the *Sofya* framework, and we show how *Sofya* improves on their shortcomings. Finally, to illustrate the flexibility and effectiveness of the framework, we describe our experiences developing several state-of-the-art dynamic analyses using *Sofya*.

1 Introduction

Many interesting problems in program verification and validation concern observing, abstracting, and modeling the behaviors programs exhibit during execution. Dynamic program analyses, which consider the execution behavior of selected program runs, are a powerful complement to traditional testing techniques. Whereas testing typically focuses on the relationship between a program's inputs and outputs, dynamic analyses are usually formulated to focus on the internal workings of a program. This allows them to detect patterns of behavior that can be abstracted and related to models, such as invariants [10], automata [2], coverage data [20, 25] or sufficient conditions for data races [27], in order to characterize program behavior, detect errors on an observed program run, or predict potential errors that may be exhibited on other program runs.

The most common application of dynamic analysis is to performance and memory profiling, but researchers have also exploited run-time monitoring to target a much wider range of program analysis applications. Broadly speaking, a

dynamic program analysis can be thought of as either a property *checker*, which compares observed program behavior to a given model, or a property *miner*, which extracts a model of observed program behavior. Runtime property checking has a long and rich history. Developed initially as a means for understanding and debugging distributed programs, over time it has evolved from checking properties of single program states, such as array bounds and null references, to checking properties that relate to multiple program locations, such as data races [27] and method atomicity [12, 32], and also to approaches that are parameterized by rich user supplied sequencing specifications [7, 13, 15, 18]. Sophisticated approaches to mining specifications have been a subject of research in recent years, resulting in tools for inferring program data invariants [10], temporal relationships between program actions [35], and object protocols [2, 34].

Implementing a dynamic program analysis for modern object-oriented languages, such as Java, presents a number of challenges. Different analysis tools address these challenges through different techniques and implementation strategies; consequently, we consider these challenges as *points of variation* in the space of possible dynamic Java analyses. We identify four major points of variation:

Program observations. Analyses may require very different information about program execution. Some analyses may be sensitive only to the occurrence of certain program events, i.e., for Java programs an event can be mapped to a class of byte-codes, whereas other analyses may require detailed information about the data state of the program. *A flexible analysis framework must support the definition of a wide-range of observations of program behavior.*

Efficient event generation. The raw material of all dynamic analyses is a stream of events that reflects the sequencing of actions and states encountered during a program run. Generating events and delivering them to the analysis implementations can slow down program execution by orders of magnitude if care is not taken. *An efficient analysis framework must build on an array of optimization strategies for minimizing the overhead of event generation.*

Selective event processing. Sophisticated program analysis problems often require that multiple sub-problems be solved and their results combined. Teasing a raw event stream apart and directing the relevant portions to each sub-problem while maintaining the independence of sub-problem implementations is a challenge in itself. Achieving this while maintaining an acceptable bound on the use of storage for analysis information is even more difficult. *A rich analysis framework must provide a generic and modular architecture for assembling complex combinations of sub-analyses.*

Concurrent event reporting. Dynamic analysis of multi-threaded Java programs is complicated by the fact that the implementation of synchronization operations is embedded within the JVM. This makes it difficult to provide high-fidelity event generation for operations such as lock acquisition and release in a manner that does not perturb thread scheduling, is portable across JVMs, and is efficient. *A dynamic analysis framework for multi-threaded programs must provide state-of-the-art strategies for generating synchronization related program events.*

To date, implementors of dynamic analysis tools have created such tools independently, with little thought given to reuse or comparability. While low-level frameworks such as the Byte Code Engineering Library (BCEL) [3] and ASM [9] provide infrastructure for implementing such efforts, they do not provide the high-level functionality required by many dynamic program analyses. Consequently, researchers have been forced to build that functionality on their own and have not, therefore, effectively leveraged each others' efforts. This not only increases the time required to implement analyses, but also the likelihood that common errors or inefficiencies will be introduced into analyses.

Beyond the problems in *creating* dynamic analysis tools, variation in implementation strategies across different dynamic analyses makes it difficult to *compare and evaluate the cost-effectiveness* of specific analysis algorithms and data structures. For example, Wang and Stoller [32] propose two very different algorithms for dynamic method atomicity analysis; if these algorithms had not been implemented in the same framework it would be difficult for researchers to empirically isolate differences in the algorithms from differences in implementation frameworks.

To address the foregoing problems, we have created (Sofya): a framework that supports the development of a wide range of sophisticated dynamic analyses for Java programs, which is available at <http://sofya.unl.edu>. Sofya offers several advantages to developers of dynamic Java program analyses. First, Sofya is designed to provide efficient and flexible solutions for each of the challenges listed above, and thus to support a broad range of dynamic analyses. Next, Sofya is architected to foster reuse of its core components as well as components that are integrated into the framework by analysis developers. Thus, Sofya offers the potential for reduced development time and run-time overhead, along with increased correctness of dynamic program analyses. Finally, Sofya provides an architecture into which multiple analysis techniques can be easily integrated. Thus, Sofya facilitates experimentation aimed at understanding the relative cost-effectiveness of dynamic analysis algorithms and data structures.

In the next section, we discuss the challenges faced when implementing dynamic analysis techniques in Java. Section 3 presents the architecture of Sofya and details the ways in which those challenges are addressed. Section 4 surveys the wide variety of analyses that are of interest to the program analysis community, and explains how their requirements are met by Sofya. Section 5 presents three example dynamic analyses from the surveyed applications that we implemented using the Sofya framework. We discuss our experiences developing these applications and quantify the effort required to build them in the context of Sofya. Finally, Section 6 summarizes the current state of Sofya, and plans for extending it.

2 Dynamic Analysis Challenges

The ability to observe program execution is fundamental to all dynamic program analyses. Events that need to be captured range from the simple, such as execution of structural entities (for example basic blocks), to the more complex, such as thread and object creation, field manipulations, and object locking behavior. Monitoring for events in Java software is most often achieved through instrumentation of byte-code, predominantly static instrumentation. However, the strict verification rules for Java class files and the stack architecture of Java byte-code can make the implementation of instrumentation difficult and error prone. Also, poorly designed instrumentation introduces performance overhead, and suffers from limitations that can affect analysis correctness, especially in a multi-threaded environment. Event capture based on the the Java Virtual Machine Tool Interface (JVMTI)¹ or Java Debug Interface (JDI) are also sometimes employed, but these too introduce challenges.

The event streams resulting from observed execution of systems are often long and complex. This can be especially problematic in multi-threaded systems where events in different threads may be arbitrarily interleaved. These issues lead to significant challenges in interpreting and storing event stream data when implementing dynamic analysis techniques. For example, trace files may be too large to store or process efficiently, or a technique might need to differentiate between events occurring in different threads.

¹Previously the Java Virtual Machine Profiling Interface (JVMPPI).

In the remainder of this section, we present detailed discussions of the challenges related to definition of program observations, byte-code instrumentation, event capture, storage, filtering and processing, especially as they pertain to multi-threaded systems.

Program observations. Specifying events in a program that are considered observations of interest is an important task in implementing a dynamic analysis. It is desirable to eliminate the dispatch of irrelevant events at the earliest opportunity to minimize overhead and maximize analysis efficiency. Ideally, an analysis implementation should constrain the instrumentation and the request of events from the JDI to the minimal amount necessary to capture the desired observations. In this way, no overhead is introduced for events that are irrelevant to the analysis. Specification of events should be clear and understandable, separating the definition of observables from details of the implementation required to generate the associated events. As the complexity of observations increase, notations for specifying them must become more expressive to support natural description of both the observations of interest and where the associated events can occur.

Existing libraries and tools such as BCEL are often used to implement instrumentation of Java class files. Such tools can meet the objective of minimizing instrumentation, but they require intimate knowledge of the construction of Java class files and the byte-code instruction set. This imposes a steep learning curve, setting the stage for the other challenges we discuss in the following sections, and leading to repetition of common errors. These tools require that program observations be defined in terms of the instrumentation implementation required to generate them (which may not be the best, or only, implementation that can be used). As a consequence, they do not satisfy the requirement that specification of observable events should be clear, understandable, and independent of implementation details.

Many analysis tools build on libraries such as BCEL to generate the program observations they require. This approach causes researchers and implementors to embed the specification of desired observations within the implementation logic of their tools. The result is that encodings of observations then become difficult to reuse for other purposes, or even to modify for new extensions to the original analysis.

As an example, consider two analysis applications (for pattern mining and property checking) that we discuss in Section 5 – both analyses require similar information, in this case about call events, but they differ with respect to where in the program they need to observe such events and what type of information they need to extract about the method calls. In a traditional approach, we would have to implement an instrumentor and event dispatch components for one analysis, then modify or create new implementations of these components customized to the second analysis.

In summary, implementors of new analyses are faced with two costly alternatives: spend the time necessary to understand and modify an existing tool used for some previous analysis (which may itself contain errors), or use a library such as BCEL to construct the analysis from scratch (and likely repeat common mistakes). To our knowledge, no tools or frameworks exist to bridge the gap between powerful yet difficult to use libraries and tools, such as BCEL and the JDI, and the task of specifying desired observations needed to implement interesting dynamic analyses.

Efficient event generation. All instrumentation has an associated cost. The extra overhead introduced at runtime is often an important consideration when performing a dynamic analysis and evaluating its usefulness. Standard practice in Java is to implement instrumentation probes as method calls to a special class, that in turn emits trace data or collects the events and produces a summarization. This is often sufficient, particularly for smaller systems and collection of infrequently occurring events. Unfortunately, for large volumes of events (such as those seen when

observing execution of structural entities) this approach can introduce intolerable overhead. Systems with frequently executing or deeply nested loops, or many threads, are especially susceptible to such overhead. For example, in an early implementation of `SoFYa` that used this strategy, a program with three-deep nested loops was observed to suffer upwards of a 250x slowdown when instrumented.

Another challenge for efficiency and correctness is how to enable observation of an entire system in a way that does not interfere with its execution environment. Specifically, there are multiple issues involved in providing for communication between instrumentation probes that are executed by the program to generate events and the analysis components that process events. One technique that was attempted in an early version of `SoFYa` is to invoke a system by executing the `main` method of the appropriate class using Java reflection. Unfortunately this technique does not result in the system running in its normal execution context, as it is now running within the same JVM as the analysis components. This forces both to compete for the same resources, such as memory, and can lead to problems in the observed system and in the analysis components (such as when the system under observation calls Java's `System.exit` method). If the observed system is executed in a separate JVM, other issues arise, such as providing efficient interprocess communication, controlling the observed system, capturing outputs, and ensuring orderly termination without loss of observed events.

Selective event processing. Dynamic analyses frequently benefit from, and often require, the ability to filter or classify runtime events occurring in the system based on type or context. For example, an analysis might need to differentiate events by their executing threads, or the identity of the objects on which they act or occur. It might even need to change its filtering criteria based on prior observed events. Techniques are frequently implemented using post-processing on monolithic traces in which actions of different threads are interleaved. A significant cost is associated with recovering the event streams for individual threads, as well as the space required to store large concurrent traces. In some cases, information about threads may even be unrecoverable, significantly reducing the precision of an analysis.

A common approach is to collect trace files and apply the analysis as a post-processing step. However, this can be impractical (and inefficient) when there are many traces to be collected or traces are very large. We have experimented with techniques that can generate multiple trace files larger than 2 GB [20] (before many of the current features of `SoFYa` were available). Most often, this is because the technique for capturing events lacks the flexibility to perform filtering and classification tasks during execution.

Different analyses may benefit from performing processing “online”, as events are received, whereas others may require the collection of trace files for post-processing. Most dynamic analyses implemented today generate a trace file in a fixed format convenient for the analysis in question, or consume events in a way that is integrated into the analysis components. Both of these approaches inhibit reuse and make such implementations unsuitable as general frameworks for implementing other analyses.

Concurrent event reporting. The greatest challenge that limits instrumentation is that of observing events in multi-threaded systems. A majority of analyses for concurrent systems are sensitive to the order of events observed in the programs, and the validity of claims derived from such analyses is greatest when the natural order of events in the program is best preserved. Thus, techniques for observing events should avoid perturbing the natural order of events in the system, yet report the ordering faithfully. Unfortunately, it is often difficult to achieve these goals with instrumentation alone.

Instrumentation involves insertion of additional executable code, which is itself executed by the threads in the program under observation. As a consequence, execution of instrumentation is subject to interruptions caused by context switches between threads, a situation that can cause the execution of a probe to be separated from the event it is intended to report. The resulting event stream will not then faithfully reflect the order of events that occurred, and in the worst case can lead to reported sequences of events that are invalid. This problem is almost always solved by protecting, with a global lock, the execution of probes and code corresponding to the event to be reported. The mutual exclusion enforced by the lock does guarantee that any event witnessed by instrumentation will occur before another event may be witnessed. However, this technique is intrusive; it disrupts the natural ordering of events in the program under observation, and incurs a severe performance penalty as the instrumented program spends much of its time contending for the lock to execute instrumentation.

As previously noted, services such as the JVMTI or JDI are sometimes utilized to avoid the problems associated with instrumentation. These services provide facilities for requesting and receiving events relevant to program execution directly from the Java virtual machine in which the observed software is running. This approach does avoid many of the problems associated with instrumentation, but it has limitations of its own. The JVMTI is a native code interface, which results in platform specific analysis tools and imposes a high learning curve on users. Both services constrain analyses to working with a limited set of events, and they are not easily extended due to their dependence on the capabilities of the virtual machine. Finally, they are targeted at performance profiling and traditional debugging, and as a consequence often lack support for finer grained selection of events, which leads to unnecessary performance degradation.

To illustrate this point, we consider the case of observing method entry events with the JDI. When these events are requested from the JDI, a discrete event is raised for every method called. A facility is provided in the JDI to filter calls, but the filtering occurs in the receiver of the JDI event stream, not the observed program. Thus every event is transmitted through slow interprocess communication channels, even if the event is not ultimately consumed. This cost is doubled if method exit events are also observed. We observed this to yield inferior performance compared to techniques that use probes to raise events only for method calls of interest.

3 Sofya Overview

We now present our framework for supporting dynamic analysis. We first describe the high level architecture of Sofya to provide a basic understanding of how typical dynamic analysis implementations make use of the framework. We then revisit the challenges described in the previous section, and discuss how various components of the framework address them. This discussion shows how Sofya both relieves practitioners of the difficulty of dealing with these challenges, and does so through abstraction of details into a clear and simple publish/subscribe architecture that facilitates rapid implementation of new analyses.

3.1 Architecture

Sofya is organized as a layered publish/subscribe architecture, illustrated in Figure 1. These layers can be grouped conceptually to identify the broader services they provide. Note that in software design, a publish/subscribe model is typically realized through use of the OBSERVER pattern, and this is the approach we took with Sofya. A component

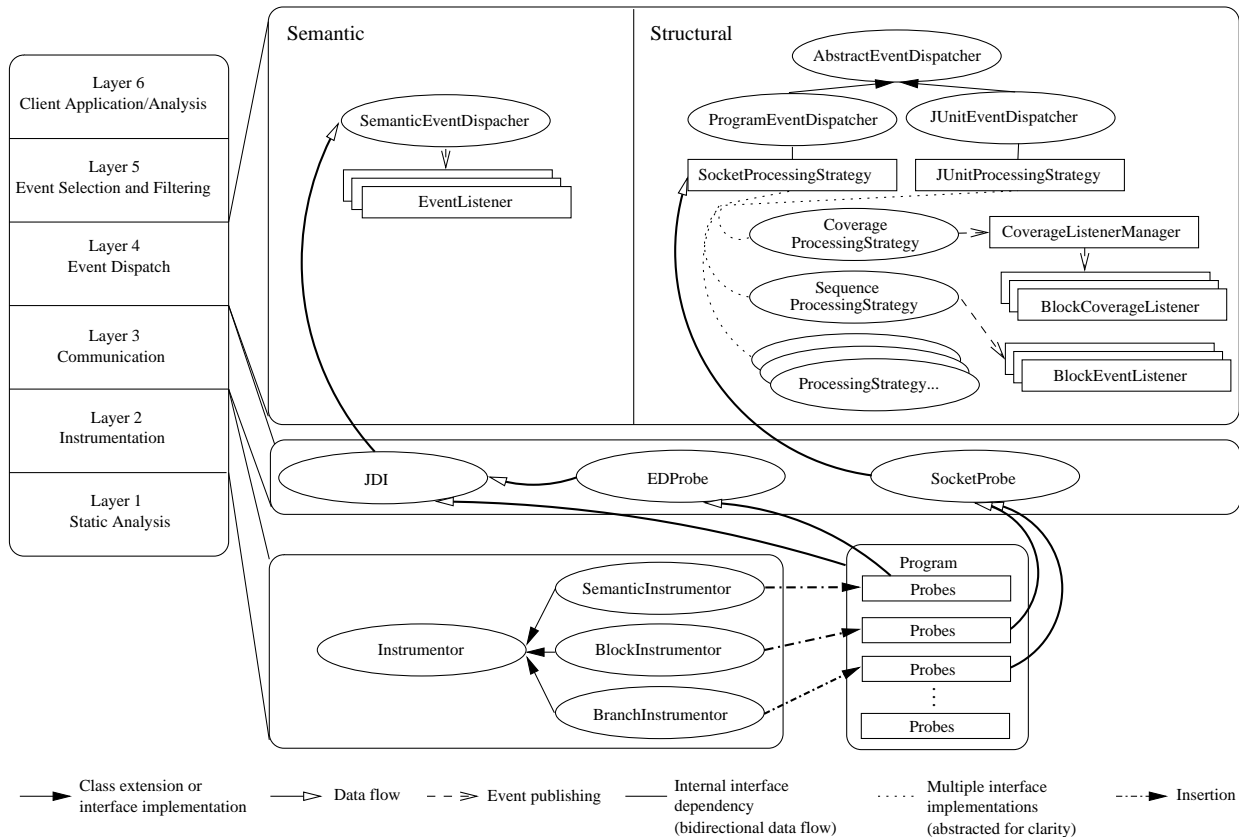


Figure 1: Sofya Layered Architecture.

that publishes events is a *dispatcher*, and a component that subscribes to events implements a *listener* interface. Thus, when referring to implementation components, these are the terms that we use. Layers 1 through 4 support the *publishing* of events to observers; Layer 4 is important as the layer at which clients may *subscribe* for events. Layer 5 supports *routing* and *filtering* of events. Layer 6 is the client application layer, where events for which clients have subscribed are consumed to implement particular analyses. We give a brief overview of each layer’s individual functions, and then discuss them in additional detail.

Layer 1 is composed of optional static analyses that provide information to guide the instrumentation process. Layer 2 handles the instrumentation of Java class byte-code to support generation of event streams. Layer 3 is the communication layer that provides transmission of instrumentation probes and JDI events from the monitored system to the event dispatchers. Layer 4 is the event dispatch layer where events are packaged and dispatched to observers; this layer defines the principle interfaces to be used by clients of the framework. Layer 5 provides classes and interfaces to support filtering and splitting event streams based on criteria such as thread and object identity. Layer 6 is purely an abstraction, to represent the layer at which analyses are implemented on top of the services provided by Layers 4 and 5.

Layer 1: The static analysis layer is used to implement analyses that are necessary to support event dispatch, or that provide guidance to other layers, especially the instrumentation layer (Layer 2). Information computed in this layer may improve the efficiency of the framework in other layers, or facilitate more precise results for some

analyses. The main implementation provided by `Sofya` at this layer is the control flow analysis supplied to the `BlockInstrumentor` (basic block) and `BranchInstrumentor` components to implement structural instrumentation and event dispatch.

Layer 2: This is the layer at which byte-code instrumentation is performed. All of the event dispatchers (at Layer 4) provided by `Sofya` depend on the use of instrumentation. Instrumentors in `Sofya` extend from the abstract base class `Instrumentor` and are built using BCEL. They may rely on information provided by Layer 1 or internalized knowledge to insert *probes* in the program to raise some or all of the events that will be dispatched in Layer 4. They are also responsible for inserting any instrumentation required to establish Layer 3 communication, using the JDI or probe classes.

Layer 3: Communication between monitored systems and the event dispatchers at Layer 4 are provided by this layer. Choosing the communications channel(s) and defining the protocols for data packaging and transmission are the primary responsibilities of this layer. The activities in this layer are often tightly integrated with Layers 2 and 4, and may be implemented to some extent in those layers. The `EDProbe` and `SocketProbe` classes provide fields and buffers used by the actual byte-code probes inserted into the program. They may explicitly manage socket communications and protocols or those functions may be provided implicitly, as is the case when using the Java Debug Interface (JDI).

Layer 4: The event dispatcher layer is the most important layer for clients of the framework, as this is the layer at which event streams are actually dispatched (published) to listeners (subscribers). Event dispatchers are responsible for accepting information received from Layer 3 and packaging it into the discrete events that are dispatched to registered listeners. `Sofya` provides event dispatchers for both semantic and structural events.

The `ProgramEventDispatcher` is the structural event publisher, which provides basic information about the execution of structural entities in the observed system, such as basic blocks and branches. It depends on components implementing a `STRATEGY` for receiving instrumentation data from the observed system, in the form of components implementing a `SocketProcessingStrategy`. As noted previously, the `SocketProbe` in Layer 3 handles processing and transmission of data, recorded to its buffers by structural instrumentation probes, to the event dispatcher using a socket. Four `SocketProcessingStrategy` implementations are provided to receive coverage and sequence execution event streams for basic blocks and branches. Coverage event streams are dispatched via the `BlockCoverageListener` and `BranchCoverageListener` interfaces, sequence event streams via the `BlockEventListener` and `BranchEventListener` interfaces. Coverage listeners are served by a listener management class, `CoverageListenerManager`, to improve efficiency in the case where only one listener is interested in the coverage event stream. Components at Layer 5 or clients at Layer 6 implement these interfaces to subscribe to event streams.

A `SemanticEventDispatcher` acts as the publisher for semantic event streams. Semantic events include, but are not limited to, events such as field reads and writes, lock acquire and release, and method call, entry and exit. Informally, we define semantic events as events that communicate information that is sensitive to the meaning of the program as it impacts data state or control dependence. The JDI provides the Layer 3 communications implementation used by the `SemanticEventDispatcher`, assisted by the `EDProbe`. A subset of the events supported for this type of event stream are implemented by the `SemanticInstrumentor` at Layer 2. Layer 1 information is specified

```

sys.prog
mod.prog
E - virtual_call FeatureImpl.* {
  in ClientManager.*
  not ClientManager.setup *
}
E + virtual_call FeatureImpl.init * { }
E + interface_call #INT RemoteAPI.* { }

```

Figure 2: Sample EDL specification.

by users of the semantic event dispatcher in a rich language for describing the events to be observed, which is described in Section 3.2. The semantic event stream is published via the `EventListener` interface.

Layer 5: This layer provides components for filtering and splitting semantic event streams published by the event dispatchers.² Of greatest interest are the thread and object stream filtering classes. The `ThreadFilter` splits an event stream, routing events occurring in separate threads to separate listeners. It depends on `ChainedEventListenerFactory` to create new thread listeners on demand. Similarly, an `ObjectFilter` class is provided to route events occurring on specific objects to separate listeners.

Layer 6: This is the layer in which client analyses are implemented as consumers of the event streams published by Layer 4 and routed by Layer 5. `Sofya` provides an atomicity checker and a regression test selection tool (see Section 5) as examples of applications at this layer. The former demonstrates direct implementation of the listener interfaces to process a semantic event stream as it is received. The latter demonstrates use of a structural event listener to generate trace files on which to perform its analysis.

3.2 Implementation Challenges Revisited

We now revisit the challenges discussed in Section 2; addressing them has been a guiding principle in the design of `Sofya`.

Program observations. `Sofya` addresses the need to define a specific set of observable events in two different ways depending on whether a structural or semantic event stream is to be published. Static input controls are provided to limit instrumentation and the selection of JDI events to the minimum necessary to publish a specified set of events. These are further augmented by runtime controls, but we save discussion of these features for when we address the need for selective event processing.

To address the need for selection of observable events when publishing a structural event stream, the instrumentors present parameters to select the types of structural entities in the program to be observed. Instrumentation is then inserted only as necessary to publish events for the selected structural entities. Observable events can also be constrained by instrumenting a subset of classes, and the programmatic APIs of the instrumentors enable selection of individual methods for instrumentation. This provides a low barrier of entry to a high level of flexibility in specifying structural events to be observed.

A rich event description language (EDL) is provided to enable the specification of observations generated in semantic event streams. EDL can specify the parts of a system on which events should be observed and published; for

²We find that applications using structural event streams rarely require complex filtering or event processing functionality.

example, the classes on which method call events should be published as program observations. A rule system supporting wildcards and an additive/subtractive precedence model supports powerful specification of subsets and specific parts of a program for which observations should be captured. The language supports even further expressiveness by providing the ability to constrain the publishing of events related to an observable based on the location at which the event is raised (for example, a call to a method on an observable class could be excluded if it occurs within a specific method of another class).

Figure 2 presents a simple EDL specification that we use to illustrate the capabilities of the language. The first two lines identify files in the `SoFya` database directory (a central repository for a variety of intermediate and output files created by and shared among `SoFya` components) that list Java classes related to the program under analysis. The first line specifies the file, “`sys.prog`”, that lists the classes that comprise the entire program, which is necessary to insert all required instrumentation. The next line specifies the file, “`mod.prog`”, that lists the classes for which program observations are published by default. This means that all events related to those classes will be published unless excluded by rules in the specification. The remainder of the file is a set of rules specifying additional constraints on what events are considered observations of interest and where they may occur.

The first rule specifies that all calls to virtual methods implemented in the class `FeatureImpl` are to be excluded from the event stream if they occur in any method in class `ClientManager` other than method “`setup`”. Package qualifiers are required for `FeatureImpl` and `ClientManager` if appropriate.

The second rule specifies that all calls to the virtual method “`init`” of class `FeatureImpl` with any signature are to be included, regardless of where they occur. This overrides the first rule for calls to method(s) “`init`”, but not for calls to other methods.

In the third rule, the “`#INT`” token is an example of an *event payload* modifier. It indicates that the method calls should be observed using an “`interceptor`” method that enables the event to provide access to information about the receiver object and arguments to the method. Because there is a higher cost associated with using interceptors, they are not used by default. However, this illustrates additional *flexibility* afforded by EDL to refine the information delivered to clients, thereby providing control over the tradeoffs between information and efficiency.

Efficient event generation. `SoFya` provides efficient instrumentation to reduce the performance penalty experienced by observed programs. When observing execution of structural entities, `SoFya` uses arrays (typically byte arrays) to record execution of code entities. Array accesses are efficient compared to method calls, and are common inside loops, which leads to very efficient handling of the instructions. Method calls are made by the instrumentation only to retrieve and commit these arrays on method entry and exit. When observing semantic events, `SoFya` writes coded integers to static fields, also an efficient operation.³ In practice, this has led to significant observed improvements in the execution times of instrumented systems. These strategies and optimizations for event generation provide a level of sophistication we believe has not been achieved by other tools for observing events in Java programs.

Layer 2 of the `SoFya` architecture offers opportunities for powerful customization in instrumentation. Unlike higher layers, customization of instrumentation may entail a greater level of implementation effort. Nonetheless, significant support is provided to facilitate such efforts, beyond that available from libraries such as `BCEL`. A new instrumentor can be implemented by sub-classing one `SoFya` class and overriding five abstract methods. The infrastructure

³The fields are monitored by the JDI, and used to then insert the appropriate events in the event stream with other events raised natively by the JDI itself.

provided by `SoFya` handles all of the details of using BCEL to load classes, provide access to byte-codes, and commit changes, in addition to a variety of utility methods to perform commonly needed but error-prone transformations.

Handling of invocation and communication between monitored systems and analysis components is greatly simplified by `SoFya`. All of the event capture components in `SoFya` provide efficient, rigorously tested implementations that handle the tasks of setting up and managing communication between observed systems and analysis components. Implementors of new dynamic analyses can focus attention on the design of their analysis tools and techniques without spending time on the details of mechanisms for capturing and communicating the events on which those analyses depend.

For applications that seek to customize the communication layer, the task is simplified to reasoning about and implementing new *protocols* for communication, rather than dealing with the details of connection and link management. This separation of concerns underscores the flexibility of the layered architecture in providing for customization without impacting other functions of the architecture significantly. It allows different protocols and communications strategies to be used while reinforcing familiarity by preserving a consistent publish/subscribe model at higher layers used by client applications.

Selective event processing. We use two approaches to address this problem in `SoFya`. First, the user has the flexibility to select events of interest and ignore irrelevant events using the features described for defining observations. This type of static control enables the exclusion of events entirely if they are known to be irrelevant to a particular analysis, with the benefit of an associated reduction in overhead. Second, controls and tools are provided to filter and classify events as they are captured and relayed to analysis tools.

`SoFya` addresses the problem of efficient event processing and storage through its use of the observer pattern. A particular analysis component that needs to work online simply implements an interface and then registers itself with the event dispatcher to receive published events. `SoFya` provides programmatic components to implement chains of filters and split events into separate streams using associated event data (such as thread identifier or receiver object). The process is transparent to attached listeners, enabling these filtered or selected event streams to be processed independently exactly as if they were the original event stream.

A trace file generator actually just becomes another filter (or “target”) at the end of a filter chain, that is processing events online and recording a trace file (or files) in whatever format is best suited to the analysis. We believe that recasting a trace file as just another type of subscriber to an event stream further illustrates the great flexibility of the layered publish/subscribe architecture. The separation of these capabilities into an independent layer supports the ability to easily implement customized event filtering and selection mechanisms, including trace file generation, facilitating rapid development of efficient analyses.

Concurrent event reporting. `SoFya` provides solutions to address issues with both instrumentation and JVMTI/JDI approaches to reliably handling capture of events in concurrent systems.

`SoFya` implements a hybrid approach combining the use of instrumentation and the JDI to publish semantic events from an observed program. Where it is not possible to obtain accurate event data non-intrusively with instrumentation, `SoFya` utilizes the JDI to capture such events. The JDI raises events synchronously with the execution of corresponding code, and guarantees the order of reported events to be consistent with the order of execution; this addresses the concerns associated with instrumentation. Instrumentation is still used where synchronization of probes with observed

events is not necessary for correctness. An example of such a situation, alluded to earlier, is a probe to observe method entry. Such a probe can be inserted as the first code in a method. If the probe has executed, the method is guaranteed to have been entered, but no program code will have yet executed. A context switch immediately after execution of such a probe has no ramifications for analyses consuming the event stream. Such a hybrid technique allows instrumentation to improve efficiency for some types of events and enables observation of custom events where safely possible, while employing the JDI to satisfy the requirement of accurate, non-interfering observation of events that cannot be handled by instrumentation alone. Based on our survey of prior work, we believe this hybrid approach is novel and represents a significant achievement in efficient and validity-preserving observation of events in concurrent systems.

Structural coverage event dispatch is inherently thread-safe. The structural sequence event dispatchers are not, an issue we will address in future work. However, we find that a majority of dynamic analyses that use structural observations are interested in coverage data, which is safely implemented using the efficient instrumentation-only scheme previously discussed.

4 Survey: Dynamic Java Analyses

Recent developments involving dynamic analysis of Java programs have involved program design, validation, verification, profiling, security, and metric collection activities, among others. We now survey a number of these analyses and the tools that implement them, and discuss how they relate to the capabilities of the `SoFya` framework. We organize this survey based on whether an analysis *mines* information from a trace and constructs a model for later use, or *checks* conformance of a trace against a given model; we note that analyses may be structured as combinations of mining and checking phases. We also discuss a common class of mining applications that is concerned with *performance* analysis.

4.1 Mining Applications

Dynamic analyses for testing and maintenance, such as coverage measurement, have been extensively investigated in the Java research and development communities. Such analyses mine program executions to accumulate information about sets of program locations; for example, whether they are reached in some program run. We describe two frameworks for building such analyses.

InsECTJ is a lightweight generic framework for instrumenting Java software [28] that offers a relatively simple and elegant design for performing basic instrumentation tasks. Its principle contribution is a framework that hides many details of using the underlying byte-code manipulation library, BCEL. Instrumentation capabilities of the type provided by *InsECTJ* are all available in `SoFya`, and custom instrumentors can also be implemented in the `SoFya` framework. Unlike *InsECTJ*, however, `SoFya` provides, as part of its core framework, existing implementations of complex instrumentation techniques that support a wide range of analyses.

The *Java Architecture for Bytecode Analysis* (JABA) [14] is a program analysis tool set that appears⁴ to provide functionality to support the kinds of analyses offered by the structural instrumentation and event dispatch components of `SoFya`, such as control flow analyses and coverage tracing. To the best of our knowledge, JABA does not provide support for more general dynamic analysis problems such as those described below.

⁴JABA is not publicly available and a detailed description of the capabilities of the tool is not available.

Neither InsECTJ nor JABA consider the challenges posed to instrumentation in a multithreaded context, and each will suffer from all of the problems described in Section 2.

Daikon [10] is a toolset for detecting likely program data invariants. Daikon mines information about data values at specific locations in a program, such as method call and return points, using the *Chicory* [6] front-end to target Java programs. Daikon’s analysis phases are independent of Chicory (to achieve language independence), and they communicate through a well-defined trace file format. Sofya could be used to implement the functionality of *Chicory* and generate such files, but one could also structure an online version of Daikon by attaching its analysis phases as listeners to the Sofya generated event stream, thereby saving the expense of writing potentially large trace files.

Recent work on mining properties about sequencing relationships between sets of program locations, perhaps distinguished by data values, has focused on inferring object protocol models for APIs [2, 34]. General approaches that attempt to *learn* arbitrary protocols have proven difficult to scale and the resulting models are hard to exploit. More recent work has looked at instantiating predefined classes of models to help with these difficulties [33, 35]. Sofya can capture all of the information needed to mine sequencing specifications that relate method calls, normal and exceptional returns, field references, and other features related to API usage. Further, Sofya can be configured to capture aspects of the data state of the program at the observed points, e.g., to correlate calls based on receiver object identity. While many tools that implement mining of sequencing specifications work offline, one could easily build online miners using Sofya, which offers the potential for eliminating large trace files. Section 5 discusses our experiences implementing Weimer’s pattern mining technique [33] in Sofya.

4.2 Checking Applications

There has been a significant amount of work using formal models of behavior, developed in the program specification and verification communities, to perform *run-time verification*. The models considered vary from built-in restricted patterns to user supplied general state machines and temporal logics. *Java PathExplorer (JPaX)* [13], *Java MultiPathExplorer (JMPaX)* [29], *JavaMOP* [5], *TGV* [15], and *HAWK* [7] are examples that range over this space of models. These tools vary in the mechanisms used to generate relevant events; for example, JPaX uses static instrumentation and JavaMOP uses aspect oriented techniques, but each decouples the checking portion to a separate consumer of the event stream and uses slightly different checking techniques, for example, various forms of automata in HAWK and JPax, and vector clock techniques in JMPaX. This kind of decoupling can be achieved in the Sofya architecture and, in fact, several analysis techniques, such as an automata checker and vector clock implementation, are available as existing Sofya components; we discuss such components in Section 5. Several of the run-time verification tools discussed above are limited in the set of observations they can make of the program; for example, properties that require information about object instance identity are in general not supported, and observing the execution of locking operations with an instrumentation-only approach, as in JPaX and Java-MOP, results in the problems discussed in Section 2. Thus, a framework like Sofya offers the possibility of supporting a broader range of such analysis problems, and of improving the accuracy and validity of analysis results.

Java-MaC is a tool that implements a run-time checking of Java programs against formal specifications [18]. Unlike the tools described above, Java-MaC employs a generative approach to insert instrumentation and create the property checking monitors it uses to check conformance. Users write specifications in a special script language that

defines the events to be captured and the properties to be verified. These are then compiled into instrumentation and monitors that are incorporated into the running program. Relative to `SoFya`'s more traditional architecture, `Java-MaC` relieves the user of the need to code analysis components, but it also makes it impossible to combine multiple analysis components to achieve sophisticated run-time analyses. Furthermore, the instrumentation in `Java-MaC` is very intrusive. It interferes with scheduling and incurs significant run-time overhead, thus reducing the fidelity of the generated analyses as described in Section 2.

In Section 5, we describe the implementation of a simple sequencing property checker in `SoFya`. We also describe the implementation of a state-of-the-art dynamic analysis for detecting method atomicity. Our implementation improves on the `Atomizer` [11] tool by incorporating supporting analyses such as dynamic race detection, escape analysis, and vector clock techniques, much as `Stoller and Wang's tool` [32] does. This provides strong evidence that `SoFya` can be used to implement sophisticated analyses.

The literature contains numerous other run-time analyses including, for example, analyses to find security flaws [22], detect concurrency anti-patterns [4], determine call chain coverage [26], and calculate object coupling metrics [1]. In all of these cases, we were able to map the application specific capabilities and components onto the `SoFya` framework.

4.3 Performance Analyses

Dynamic Java analysis tools that perform profiling and performance data collection appear to be in wide use; we found 18 active Sourceforge projects focused on such tools. These applications range in capability from simply accumulating information about the number of occurrences of specific events (for example, calls to a specific method), to gathering information about resource utilization, space consumed by instances of a type, garbage collection information, or run-time of method activation.

Most Java profiling tools use the Java Virtual Machine Profiling Interface (JVMPi) to capture program execution events that are used to compile dynamic performance metrics. The JVMPi is a lower layer interface to the same types of events available through the JDI; thus, the same types of events can be captured using `SoFya`. `SoFya` is not designed explicitly to support performance analysis applications, but any profiling application that simply accumulates counts of events such as method invocations, object allocations, or synchronization statements is a simple `SoFya` application. `SoFya` could easily be extended to support a broader range of performance analyses; for example, the delivery of timestamps with events would allow most of the timing-related capabilities available in existing Java tools to be implemented as `SoFya` clients, and adding optional timestamp payloads is a trivial extension to `SoFya`'s existing event-generation support.

5 Sofya Applications

To evaluate the utility, flexibility, and usability of `SoFya` we selected three analysis applications from the set surveyed in Section 4 to implement in `SoFya` ourselves. The first two analyses were implemented by the first author, who is the developer of `SoFya`. The final analysis was implemented by the second author, who was not familiar with the architectural details or APIs of `SoFya`. We report information from personal time-logs that we maintained during the development of the two smaller analyses. While anecdotal, we believe that this information on development

effort provides evidence that `Sofya` facilitates rapid development of interesting analyses by both experienced and inexperienced users of the framework.

The first application we implemented, which we discuss in the greatest detail, is a version of the reduction-based dynamic atomicity checking algorithm described by Wang and Stoller [32]. This application is composed of a set of core components to implement the logic of reduction automata, and a set of supplementary analysis components described in [32]. These supplementary analyses are implemented to improve the precision of the atomicity checker; however, the modular design facilitated by `Sofya` enables them to function as independent analysis components that can also be used by other analyses built on the framework. The second application is a simple technique for mining $(ab)^*$ patterns from program traces presented by Weimer and Necula [33]. The third application is a finite state automata (FSA) property checking technique that utilizes the Propel [31] framework from the University of Massachusetts. We conclude this section with a brief description of several other dynamic (and static) analyses that are implemented in or supported by `Sofya`.

5.1 Reduction-based Atomicity Checking

Lipton [21] developed the concept of *reduction* as a means for simplifying reasoning about the correctness of parallel programs. The key concept in reduction is to determine whether the effects of all activations of a given method in all parallel program executions are equivalent to the effects of some sequential activation of the method; such a method is said to be *atomic*. Intuitively, method atomicity captures the informal notion of *thread safety* that is often a goal in developing Java components. Method atomicity for Java has garnered interest in recent years due to work by Flanagan and colleagues [11, 12] in developing both static and dynamic analyses to reason about method atomicity.

We focus on the recent work of Wang and Stoller [32], who have developed a state-of-the-art dynamic analysis for method atomicity. More specifically, we focus on their reduction-based analysis. The idea behind all reduction-based reasoning is that all actions performed by a method can be classified as one of four types of operations. Using this classification scheme, if the sequence of actions in a method satisfies a particular regular expression, the method is judged to be atomic. Abstractly the regular expression enforces two requirements on the execution of a method: (1) the method cannot be involved in any data races, and (2) the method must be able to run to completion without requiring execution of another thread; we refer the reader to [32] for details.

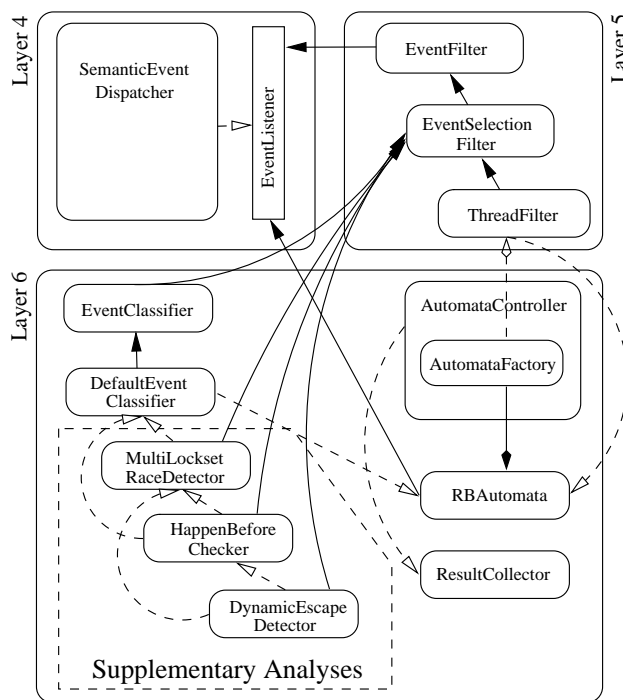


Figure 3: Atomicity Checker Architecture

5.1.1 Implementation

Figure 3 presents the architecture of `Sofya`'s atomicity checker; solid edges with triangular heads indicate interface implementation or class extension, solid edges with diamond head indicate instance creation, and directed dashed edges indicate data flow. Conceptually, the checker can be divided into its *core* components and a set of *supplementary analyses* that are used to boost precision.

The core components are responsible for classifying observed program events and matching event streams for methods to the regular expression indicating atomicity. An `EventClassifier` maps events received from the `SemanticEventDispatcher` as one of four symbols, denoting the classifications in the alphabet of the regular expression. An `RBAutomata` class instance matches symbols from the event stream to the regular expression for a method activation. The `AutomataController` observes the event stream, starting and stopping individual automata on method entry and exit. Events are forwarded to active `RBAutomata`, which request classification from the `EventClassifier`, or in certain cases classify the events themselves to drive state transitions. A `ResultCollector` merges the final states of all automata executed for each method. If all instances of the automata for a given method reached an accept state, the method is reported as atomic.

These components are configured and connected by a main method (not shown) that consists mostly of standard code used by analyses implemented on the semantic event dispatch components. The application specific parts of this involve creating a `DefaultEventClassifier`, which classifies events using simple heuristics, and an `AutomataController`, and registering them as `EventListeners` with the `SemanticEventDispatcher`.

5.1.2 Supplementary Analyses

Even a basic implementation of the reduction-based automata strategy for dynamic atomicity checking effectively reveals atomicity errors in programs [11]. Wang and Stoller discuss three supporting dynamic analyses, each of which can improve the precision of the analysis to reduce the rate of false negative reports (that is, atomic methods reported as being non-atomic). The first author also implemented versions of these supporting analyses, and we overview their implementations below; we defer a number of details of the implementations to their documentation, available at <http://sofya.unl.edu>.

Each analysis is implemented as an independent component in `Sofya`. They are combined with the core analysis, using `Sofya`'s event-listener framework, to produce an atomicity analysis that is comparable in terms of precision and performance to the one described in [32]. We believe that the ability to create these different analyses, each of which is interesting in its own right, and make them available as building blocks for other analyses built using `Sofya`, illustrates the ability of `Sofya`'s architecture to promote reuse and thereby reduce development effort.

Dynamic Escape Analysis: most interesting data manipulated by Java programs is stored on the heap. The default atomicity classification pessimistically assumes that all accesses to heap data are visible to multiple threads, but if one can determine that a heap-access influences a single thread it can be classified differently, improving analysis precision. We implemented the dynamic escape detection algorithm described in [32], which considers an object escaped if it is assigned to a static field or already escaped object, passed as an argument to a native method, or used as the runnable target of a new thread. Our `DynamicEscapeDetector` analysis component subscribes to receive events from the `SemanticEventDispatcher` and monitors for these conditions. The escape detector uses unique

identifiers, assigned by the framework to each observed object, to record escaped objects and provide a query method to check whether the object with a given ID is escaped.

Multi-lockset Analysis: a heap object that is judged to escape a thread is not necessarily accessible to multiple threads concurrently. Multi-threaded Java programs commonly use a locking discipline to ensure coherent access to shared heap structures. The lockset algorithm attempts to determine whether any concurrent read and write accesses to a field are not protected by a common lock. It is sufficient to note that a lockset algorithm can be implemented if it is possible to determine the locks held by a thread when it accesses or modifies a field. Locks in Java are built into each object; thus, `Sofya` can uniquely identify each lock using the object identifier. For any event, `Sofya` is able to report the locks held by the thread at the time of the event, which allows the `MultiLocksetRaceDetector` analysis to be implemented as a subscriber to the semantic event stream.

Happens-before Analysis: just because a heap object is not lock-protected does not mean that it is accessed by multiple threads concurrently. Synchronization between threads that is unrelated to the shared heap object may imply a strict ordering on accesses from those threads. A happens-before analysis attempts to determine whether two accesses to a field involving at least one write access can be concurrent. The analysis uses information about when threads start and join each other to decide whether field accesses are possibly concurrent, and can be extended to account for ordering imposed by the use of lock waits and notifies.

Wang and Stoller describe a technique for implementing a happens-before analysis using directed acyclic graphs, but they point out that a more efficient implementation using vector clocks may be possible [32]. We implemented this improvement by adapting O’Callahan and Choi’s [23] hybrid limited happens-before algorithm using vector clocks. The `HappensBeforeChecker` analysis is implemented as a subscriber to the semantic event stream. Vector clocks are created and updated on each thread start or join event. On each field access, the vector clocks for live threads are associated with that field. It is then possible to provide a method to query whether a field access is potentially concurrent with any access to the same field.

5.2 Mining Temporal Specifications

Weimer and Necula [33] presented a heuristic for mining simple temporal patterns of the form $(ab)^*$; that is, patterns that specify that a actions and b actions occur in matching pairs. The classic example of such a pattern is the intended sequencing of `open` and `close` operations on a file. Their observation was that the b events are likely to be represented in exception handling code. If such a pattern exists, its presence will be attested to by the programmer’s concern in assuring that the b always gets executed, even in the face of exceptional program behavior. For the purposes of this analysis, candidate events are always method calls.

Mining for these patterns using this heuristic involves three steps. First, the analysis must identify method calls that are in exception handling sections of the code. We built a simple parser, using the parser generator ANTLR [24] and a grammar for Java 1.5 source code [16]. The grammar was instrumented to identify calls occurring in `catch` and `finally` blocks, so that the parser can output a file containing a list of such calls. Second, we created a `Sofya` application that inputs the file produced by the parser, a semantic event specification that activates observation of calls, and the program to be observed, and uses the semantic event dispatcher to run the program. Call events are dispatched to an analysis component that uses the list of “cleanup” calls to build a set of candidate ab patterns. In the third

Analysis	Developer	Learn Sofya	Learn Other	Design +Impl.	Total	LOC [17]
(ab)*	lead dev.	0:00	2:08	6:12	8:20	223
FSA	2nd author	2:30	1:53	1:58	6:25	160
OSFSA	2nd author	0:32	0:00	0:33	1:05	149

Table 1: Analysis Development Time (hours:minutes)

step, heuristics are used to screen the set of patterns reported to users; we did not implement the screening techniques described in the paper, as they are not central to assessing the value of Sofya in implementing this analysis.

Table 1 reports the time recorded by the *lead* developer of Sofya to build the (ab)* analysis.⁵ The time is broken down into time spent *learning Sofya*, which for the lead developer was 0, time spent *learning other* tools, which in this case was ANTLR, *design and implementation* time, and *total* time. The final column in the table reports non-comment source lines of code for the Sofya related portions of the analysis implementations.

5.3 FSA Property Checking with Propel

Tools for checking conformance of execution traces with finite-state automata (FSA) specifications have been developed by a number of researchers. Such tools require support for creating the specification, mapping the transitions automata onto program actions, tracking the state of the automata along a program execution, and rendering a verdict about conformance.

We used the Propel [31] tool to create FSA specifications. Propel has many advantages over existing approaches for creating such specifications, and it makes use of a rich underlying framework for creating, manipulating, and serializing finite-state automata. In Propel we can define, for example, an alphabet of symbols of the form `ClassName:MethodName` that are used to construct specifications of legal sequences of API calls; these symbols are also used to generate EDL for Sofya. Once we have defined the structure of the desired call sequences, Propel encodes those sequences as an FSA.

We implemented two versions of a dynamic FSA conformance checker in Sofya: *per-thread* and *per-receiver-object* checkers; we refer to the latter as an *object-sensitive* FSA checker (OSFSA). Both checkers implement a sub-type of `EventSelectionFilter` that handles calls on the methods and classes encoded in the EDL and triggers the appropriate transitions in the FSA; each instance of this sub-type stores the current FSA state reached by the sequence of calls.

The per-thread checker applies a `ThreadFilter` to the raw event stream to split the stream into separate event streams for each active thread in the system. An `EventSelectionFilter` instance is generated for each thread and chained to the output of the thread filter; thus, the sequencing constraint of the FSA is checked against the calls produced by each thread separately. An error is indicated if any thread’s call sequence ends in a non-accepting FSA state. The object-sensitive checker applies an `ObjectFilter` to the raw event stream to split the stream into separate event streams for each distinct allocated object in the system. By minimizing the EDL specification, only instances of classes named in the FSA alphabet will ever have events generated for them. As above, an error is indicated if

⁵We do not report a time log for the atomicity application because it was initially built on an older version of the Sofya API that experienced changes concurrent with its development.

the sequence of calls on any object instance ends in a non-accepting FSA state. The resulting checker is capable of detecting API usage errors arising from improper synchronization.

The FSA and OSFSA analyses were implemented by the second author of the paper. Previously, this author had looked at neither the `Sofya` nor the `Propel` APIs. Furthermore, the author’s understanding of the architecture of `Sofya` at that time was probably less well-developed than what a careful reader of this paper will garner. As one would expect, this developer had to spend a non-trivial amount of time learning the `Sofya` APIs.

Due in part to the fact that the FSA checker is a fundamentally simpler analysis than the $(ab)^*$ miner, and to the fact that the author was able to use the $(ab)^*$ miner as a model for implementation, the development time for this analysis was quite small. The additional time needed to develop the OSFSA checker was very short, indicating that knowledge of `Sofya`’s architecture and components can allow analysis variants to be developed very quickly.

5.4 Overview of Other Applications

A variety of other applications and analyses have been developed, a number of which use the structural event dispatch components of `Sofya`, some of which we provide with `Sofya`. Most originate as applications built on `Galileo`, an older tool set from which `Sofya` evolved.

`DejaVu` is a tool for regression test selection in Java. Test selection uses change information between versions of a software system to select test cases that exercise those changes. This is accomplished by executing the test cases for a version of the system and recording the structural coverage achieved by those test cases. Coverage information from the test suite is used to determine which test cases cover changed code and those test cases are selected [25]. `DejaVu` uses the basic block instrumentation and event dispatch components, thus serving as an example of a sophisticated testing technique implemented using the structural event dispatch capabilities of `Sofya`.

Structural event dispatch components of `Sofya` have been used to implement and evaluate test case prioritization techniques [8] and dynamic impact analysis techniques [20]. Test case prioritization is another widely investigated regression testing technique that seeks to reorder test suites such that test cases with the highest probability of exposing faults are executed first. Dynamic impact analysis, which uses the structural sequence tracing capabilities of `Sofya`, seeks to determine and report how changes to one part of a program impact other parts of the same program.

Finally, `Sofya` provides implementations of advanced static type inference techniques for exceptional control flow [19], based on interpretations of algorithms for varying the costs of analyses against the precision of resulting control flow representations [30]. This offers advanced capabilities in Layer 1 that allow clients of the structural event dispatch components of the framework to choose tradeoffs between analysis cost and precision. To the authors’ knowledge, these are the only publicly available implementations of these algorithms, offering the possibility of precision in control flow based analyses not achievable in other tool sets.

6 Conclusions and Future Work

Dynamic analysis tools and techniques are important to the practice of software engineering. They are applied to a wide variety of problems, and improve practitioners’ confidence in the correctness, security, performance, and reliability of software systems. They are also rapidly gaining importance for their ability to contribute to the solution of hard problems in static analysis, and vice versa. Thus it becomes ever more important that the community have access

to a framework to facilitate the implementation, evaluation, and comparison of new dynamic analysis techniques – a framework that relieves researchers and practitioners of the challenges inherent in constructing robust, reliable dynamic analysis tools for Java software, including multi-threaded software. We believe that Sofya provides just such a framework for enhancing and driving forward the state of the art in dynamic program analysis in software engineering. Sofya addresses these challenges with an architecture that abstracts the solutions behind expressive interfaces, and we have demonstrated the effectiveness of the framework with example applications.

There are areas of extension to Sofya that we intend to pursue in future work. We would particularly like to pursue avenues for dynamic instrumentation removal, where such a capability is appropriate, and new additions to the Java language hold strong promise in this area. As the JDI continues to evolve with new releases, we will continue to explore ways to exploit this functionality to support new, and improved, handling of events. Such changes will improve the flexibility and performance of the framework.

We are making Sofya available to the public, with examples, tutorials, and extensive API documentation. Current information on Sofya can be found at <http://sofya.unl.edu>. It is our hope that this will facilitate rapid development of new dynamic analysis tools and techniques, and encourage more frequent and reliable empirical investigation and comparison of such tools and techniques. By enabling researchers and practitioners to more readily consider evaluation and comparison of proposed techniques, we believe this will accelerate advances in the state of the art in dynamic program analysis and thus enable more aggressive delivery of new techniques that will improve the reliability of all software.

Acknowledgments

This work was supported in part by the NSF through awards 0429149, 0444167 and 0454203. We thank Scott Stoller, Liqiang Wang, and Westley Weimer for answering questions about the analyses they built. We thank Rachel Cobleigh, Ricky Chang and Nathan Jokel for answering questions about Propel and the UMass FSA package.

References

- [1] Áine Mitchell and J. F. Power. An empirical investigation into the dimensions of run-time coupling in Java programs. In *Int'l. Symp. Princ. Pract. Prog. Java*, pages 9–14, 2004.
- [2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Symp. Princ. Prog. Lang.*, 2002.
- [3] <http://jakarta.apache.org/bcel>.
- [4] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of Java applications for multithreaded antipatterns. In *Int'l. W. Dyn. Anal.*, 2005.
- [5] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Int'l. Conf. Tools Alg. Const. Anal. Sys.*, LNCS, 2005.
- [6] <http://pag.csail.mit.edu/daikon/download/doc/daikon.html#Chicory>.

- [7] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Int'l. W. Dyn. Anal.*, 2005.
- [8] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Emp. Softw. Eng.*, 11(1):33–70, 2006.
- [9] Éric Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *J. Composants 2002 : Systèmes à composants adaptables et extensibles*, Nov. 2002.
- [10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.
- [11] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symp. Princ. Prog. Lang.*, pages 256–267, 2004.
- [12] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conf. on Prog. Lang. Design Impl.*, 2003.
- [13] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Meth. Sys. Design*, 24(2):189–215, 2004.
- [14] <http://gamma.cc.gatech.edu/jaba.html>.
- [15] C. Jard and T. Jeron. TGV: Theory, principles and algorithms. *Int'l. J. Softw. Tools Tech. Trans.*, 7(4):297–315, Aug 2005.
- [16] <http://wwwantlr.org/grammar/1090713067533/index.html>.
- [17] <http://www.kclee.de/clemens/java/javancss>.
- [18] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Meth. Sys. Design*, 24(2):129–155, 2004.
- [19] A. Kinneer and G. Rothermel. Assessing the Cost-Benefits of Using Type Inference Algorithms to Improve the Representation of Exceptional Control Flow in Java. Technical Report TR-UNL-CSE-2005-0002, University of Nebraska - Lincoln, May 2005.
- [20] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Int'l. Conf. Softw. Eng.*, 2003.
- [21] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Comm. ACM*, 18(12):717–721, 1975.
- [22] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Conf. O.O. Prog., Sys., Lang., and Appl.*, pages 365–383, 2005.
- [23] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symp. Princ. Prac. Par. Prog.*, 2003.
- [24] T. Parr and R. Quong. ANTLR: A predicated LL(k) parser generator. *Soft. Prac. Exp.*, 25:789, July 1995.

- [25] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, 1997.
- [26] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *Int'l. Symp. Softw. Test. Anal.*, pages 1–11, 2004.
- [27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Sys.*, 15(4):391–411, 1997.
- [28] A. Seesing and A. Orso. InsECTJ: A generic instrumentation framework for collecting dynamic information within Eclipse. In *W. Eclipse Tech. eXchange*, 2005.
- [29] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Symp. Found. Softw. Eng.*, pages 337–346, 2003.
- [30] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.
- [31] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: An approach supporting property elucidation. In *Int'l. Conf. Softw. Eng.*, pages 11–21, 2002.
- [32] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, Feb 2006.
- [33] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Conf. Tools Alg. Constr. Anal. Sys.*, pages 461–476, April 2005.
- [34] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Int'l. Symp. Softw. Test. Anal.*, pages 218–228, 2002.
- [35] J. Yang, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Int'l. Conf. Softw. Eng.*, 2006 (to appear).