RC23364 (W0410-043) October 7, 2004
Computer Science

IBM Research Report

SABER: Smart Analysis Based Error Reduction

Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, Larry Koved
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

SABER: Smart Analysis Based Error Reduction

Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan,
Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, Larry Koved
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY, USA 10532
(914) 784 7923
dreimer@us.ibm.com

ABSTRACT

In this paper, we present an approach to automatically detect high impact coding errors in large Java applications which use frameworks. These high impact errors cause serious performance degradation and outages in real world production environments, are very time-consuming to detect, and potentially cost businesses thousands of dollars. Based on three years experience working with IBM customer production systems, we have identified over 400 high impact coding patterns, from which we have been able to distill a small set of pattern detection algorithms. These algorithms use deep static analysis, thus moving problem detection earlier in the development cycle from production to development. Additionally, we have developed an automatic false positive filtering mechanism based on domain specific knowledge to achieve a level of usability acceptable to IBM field engineers. Our approach also provides necessary contextual information around the sources of the problems to help in problem remediation. We outline how our approach to problem determination can be extended to multiple programming models and domains. We have implemented this problem determination approach in the SABER tool and have used it successfully to detect many serious code defects in several large commercial applications. This paper shows results from 11 such applications that had a total of 455 coding defects.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software Program/Verification – *validation.*

General Terms
Verification, Performance, Languages.

Keywords
Frameworks, Defect Understanding, Program analysis.

1. INTRODUCTION

Frameworks such as J2EE™ are powerful [3] because they provide standardized components and abstractions that address common application needs and allow users to build more sophisticated applications. Like most other powerful and complex frameworks, J2EE can have negative implications for applications if used incorrectly. The problems of greatest impact are those that are not easily found during development and testing, but manifest themselves after deployment under the stresses of a production environment. Resulting production outages cost businesses thousands of dollars, require highly skilled personnel to find the problems, and sometimes take weeks or months to find.

The results presented in this paper are based on three years of experience solving critical customer problems in large scale deployments of transactional Java (J2EE) applications. We observed the following classes of recurring problems:

Resource Management: Improper cleanup of resources such as database connections, Input and Output streams, files, sockets, HttpSessions, EJBs etc. on any program execution path can result in system instability, because the system will eventually run out of these resources, as the load on the system increases.

Concurrency: Concurrency related bugs, such as data races due to unsafe updates to shared data (e.g., servlet or static fields) or calls to external resources from synchronized blocks are usually extremely hard to identify in production.

Server side Java: When Java is used for server side J2EE applications that are intended to run for long periods of time, popularly referred to as 24x7 applications, careful attention is important to avoid expensive calls and unnecessary serialization. Failure to do so can result in scalability problems and system outages in production (e.g., inability to service requests because threads are either unnecessarily suspended, or because the system is performing unnecessary garbage collection).

Persistent Data Management: In applications that are transactional, it is important to pay attention to details of persistent data. For example, storing non-serializable objects in other objects which are persisted can result in loss of information. This type of event occurs usually under load, and is thus difficult to diagnose.

™ Java 2 Enterprise Edition (J2EE) is a trademark of Sun Microsystems.

Implementation Contract Violation: Violation of the implementation contract of methods that are informal and not explicitly defined in the framework specification, *equals()-hashCode()* for example, can result in incorrect application behavior which is extremely hard to diagnose in production.

The detection of such problems in production is time consuming and expensive because they surface only under load, or appear intermittently. It is therefore important to detect these problematic coding patterns during development, well before the application is deployed in production.

Typically, the types of defective coding patterns discussed above span multiple classes and methods, and they require the identification of specific call sequences and object instances. For example, to detect if a resource is closed on all paths in the program, we need to identify the call sequence starting from the creation of the resource to all possible program exit points. Further, for all calls that close a resource, we need to ensure that the resource object instance that is currently being tracked is closed. Detection of such defective coding patterns can be done statically or dynamically. This paper describes SABER, a tool which uses the results of interprocedural static analysis of the application to detect problematic coding patterns.

In addition to a static analysis engine, a tool for detecting large numbers of problematic coding patterns requires a language or some other extensibility mechanism for describing defective coding patterns. The design points for SABER also include scalability and usability. We rejected using a very general language for expressing bad coding patterns because of these latter two goals. The iterative application of static analysis for the detection of several hundred general coding patterns is prohibitively expensive. Also, we were concerned that our target users, field engineers, would not learn a new language. Instead, we classified the patterns into a handful of categories, based on the similarity in algorithms needed for the detection of these patterns. This classification allows us to have one optimized detection algorithm for each category, and one easily understood "rule" for each category. This approach has two important benefits (a) it allows the tool to scale to large applications (the largest application currently analyzed by SABER has 8770 classes) and (b) it allows the tool to be customizable and extensible to new coding patterns as long as they fall into the broad categories.

One implication of using static program analysis for the detection of these coding patterns is that it can result in a large number of false positives, specifically, claims that a program exhibits a certain behavior that in fact could never occur. False positives can seriously limit the usability of a tool, because of the sheer amount of time a user needs to invest to differentiate the real coding errors from the false positives. We address the issue of false positives by introducing

automatic ***filtering*** mechanisms within the tool. These mechanisms filter out those false positives by incorporating domain knowledge into the specification of the coding pattern. This knowledge is incorporated into an external specification of the coding pattern, which allows the filtering mechanism to be extensible as well (i.e., new filters can be easily added to each pattern specification).

The above approach eliminates the majority of false positives, but a smaller set of false positives can only be filtered manually with application specific knowledge by the user. An important implication of such filtering is that the tool needs to provide the user with sufficient program context to help either rule out the pattern as a coding defect (i.e., classify it as a false positive), or to help the user understand how to correct the defect. Because the defects are typically subtle and span multiple program components, we provide different mechanisms to help in problem remediation.

Contributions: SABER is similar to both xgcc [12][16] and ESP [4], where users may specify rules representing bad code patterns, and a static analysis engine is used to find specific occurrences of these patterns in code. These systems analyze C and C++ programs, while SABER analyzes Java. The new contributions of SABER are:

- We describe J2EE framework domain constraints, and the common problems (coding patterns) that we have encountered in large commercial applications which result from the lack of constraint enforcement.
- We classify the sources of these problematic coding patterns into a small set of categories, based on the kinds of program analysis necessary to detect these coding patterns in application code.
- We describe how to combine domain specific knowledge with results from static program analysis to identify code defects in large applications.
- We incorporate domain-knowledge-based false positive filtering mechanisms in the tool to automatically filter out a class of false positives that require domain knowledge.
- We introduce explanation and exploration mechanisms to help the user in filtering and in understanding defects, and explain the heuristics we use to select the display of contextual information to the user.
- We describe how to tailor static analysis to effectively and efficiently analyze J2EE applications.
- We describe how we extended SABER to other problem domains.

The rest of the paper is organized as follows: Section 2 discusses the role of framework domain knowledge in problem determination, and gives examples of the domain knowledge used in SABER. Section 3 outlines the categorization of problematic coding patterns in application code. Section 4 describes our approach to filtering false

positives. Section 5 provides a description of the SABER architecture. Section 6 discusses the SABER rule analysis approach. Section 7 describes static analysis considerations for J2EE semantics. Section 8 talks about program defect understanding in SABER. Section 9 validates the performance of SABER on several large commercial J2EE applications. Section 10 discusses extensions to the tool to detect problematic coding patterns in a different domain. Section 11 discusses related work and Section 12 concludes the paper.

2. FRAMEWORK DOMAIN KNOWLEDGE

Frameworks provide software layers on top of the core Java language, making application programming much easier by providing common functionality and services for a given domain. For example, for the GUI domain, frameworks provide widgets, event handling, and models. For the e-commerce server domain, frameworks provide support for transactions, session management, resource pooling, security, and availability. Middleware layers, often referred to as a programming model "stack", are illustrated in Figure 1. Several J2EE frameworks, EJB (Entitiy Java Beans) and Web Services, are layered over the core Java J2SE packages [19].

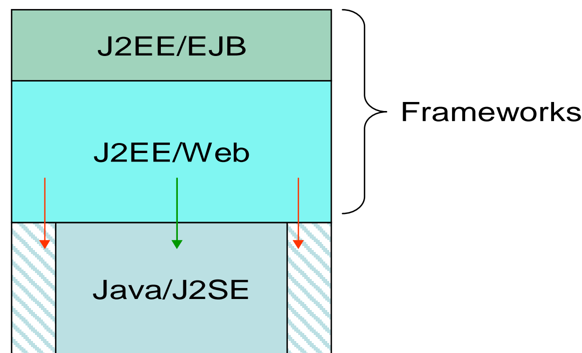

Figure 1. Programming Model Stack

However, while vastly simplifying application development, frameworks introduce both explicit and implicit constraints in the use of the generic programming model which are not enforced. Thus, the generic J2SE programming model that is legal in a general context is restricted in the J2EE [20] context. Application programmers must be experts in J2EE usage to avoid subtle runtime problems. Many of the J2EE constraints derive from the fact that J2EE provides a

collection of transaction-based services which must be highly scalable and available. Examples of J2EE domain knowledge that the skilled programmer needs to know are described in the following subsections.

2.1 Thread-safe Servlets

A J2EE application server uses multiple worker threads to service HTTP requests. Generally, each type of HTTP request is serviced by a single *Servlet* object, which is shared by all of the worker threads. Thus, certain methods of any object derived from the *Servlet* class (e.g. *service(), doGet()* or *doPut()*) must be thread-safe. All methods called from these specific *Servlet* methods are similarly multi-threaded. Consequently, *Servlet* instance fields, in addition to static fields, must be treated as shared variables. In fact, it is better not to modify these kinds of fields at all in transactions, since accessing shared, writable fields requires the use of critical sections, and critical sections can create bottlenecks which reduce scalability. Well-written J2EE applications use few critical sections, relying on the underlying frameworks to manage resources.

It is possible to force *Servlets* to be single-threaded by implementing the *SingleThreadModel*, which results in a single *Servlet* object for each thread. However, this practice is not advised for performance reasons. Potentially, too many threads will be created.

2.2 EJB Data Integrity

EJBs are Java beans that provide safe and scalable access to persistent data in transactions. Since EJBs are designed to be used transparently in a distributed and multi-threaded environment, files and other modifiable shared state (e.g. statics which can change in value) must not be accessed in an EJB.

2.3 Threads

Best practice dictates that threads should not be started in *Servlets*. Introducing multiple threads increases the concurrency and complexity of the application, causes hard-to-find bugs, and potentially reduces scalability, due to increased synchronization. Work within a *Servlet* should be done serially—parallelism is achieved by the application server's use of multiple worker threads. Additionally, the J2EE specification states that threads should never be manipulated in EJBs.

2.4 Session Invalidation

The *HttpSession* class is provided to save state across the multiple web requests that comprise a single user session. A single *HttpSession* object is typically created for each user session. By default, there is a time-out mechanism for closing sessions and deallocating *HttpSession* objects. However, since the timeout period can be very long (several hours), each session should be invalidated as soon as it is no longer needed, by calling the session's *invalidate* method. Typically this is done as part of a user logoff operation. By invalidating sessions as soon as possible and not waiting for them to time out, caching behavior is optimized, and the overall memory used by the application is reduced.

2.5 Persistent Objects

All objects stored directly or indirectly in an *HttpSession* may be written out to disk depending on choices made at application deployment time and system load. A developer must ensure all objects stored in an *HttpSession* implement the *Serializable* interface. If an object stored in an *HttpSession* object does not implement the *Serializable* interface, it will not be available when the application tries to read it back in from disk.

In one critical performance situation, the customer experienced increased system load which resulted in a large percentage of web-based users of the system receiving error pages. After several days of analyzing application logs and generated exception information, we discovered that as the workload increased, the *HttpSession* information no longer fit in the in-memory cache, and had to be written to persistent store. When these *HttpSessions* were read back into memory, the objects stored in them which did not implement *Serializable* were null. This resulted in the error pages being generated for these users. Since this problem only showed up intermittently under load, it took several highly skilled people close to two weeks to find and fix it.

2.6 Expensive or Inappropriate Operations

Expensive operations should not be performed within a *Servlet* or EJB. For example, using the JAR utility in a *Servlet* will greatly limit performance and scalability. Reflection should also be avoided from *Servlets*, because of performance and understandability. It is better to use explicit interfaces.

Certain functionality does not make sense on the server. For example, any use of *sound* in server code should be flagged. Sometimes programmers try to implement functionality which is better left to an underlying framework. For example, calling any methods in the *Runtime* class indicates that the programmer is trying to manage memory, such as

garbage collection, or some other system-level resource, which is inappropriate for a J2EE application, and can result in degraded performance.

2.7 Resource Management

It is important that connections to databases are closed on all possible application paths and as soon after they are used as possible. We discovered in a consulting engagement for a large financial application, after several days of debugging, that some database connections were being closed only in the *finalize()* method of an associated object. Because the *finalize()* method is only called during garbage collection in Java, database connections were held longer than necessary resulting in potential database connection leaks. In production, we spent a significant amount of time trying to reproduce the problem, because only the right combination of datasets and system load could reproduce the problem. This problem was solved only after several days of analyzing a number of Java virtual machines' traces in combination with the performance data from the rest of the system.

2.8 Expensive Operations in *finalize*

More generally, since *finalize()* is run only when garbage collection occurs, any operation which is CPU intensive or which can block should not be called from *finalize()*. Expensive finalize processing can create a serious bottleneck for all the worker threads. Instead, a cleanup method which is explicitly invoked should be used.

In another consulting engagement, in a large financial institution's production system the method *System.gc()* was being invoked in the *finalize()* method of a third party framework component. The manifestation of this problem in production was that the system spent only approximately half its time processing application requests as the other half of its time was in garbage collection. Since the actual load on the application was much greater than the expected workload, previous testing had not uncovered this problem and the existing test environment could not handle the necessary load to reproduce the problem. This meant that the problems would have to be analyzed in the production environment. After several days of debugging with the limited information and limited changes that could be introduced in the production environment, we found that a single memory allocation failure would be immediately followed by as many as 60 to 80 direct garbage collection cycles. To try to find out where garbage collection was being invoked, we instrumented *System.gc()* itself and found that *System.gc()* was being invoked by a *finalize()* method in the third party framework code.

2.9 Method Contracts

Some method pairs in different classes are closely related, so that when one method is over-ridden, the other method must be over-ridden also. A frequent error is to override the *equals()* method of a class, without similarly overriding the *hashCode()* method. As a consequence, for example, two objects which are equal according to the new *equals()* method may end up with different hash codes. Similarly, *readObject()* and *writeObject()* must be overridden consistently. Note that this type of error occurs in general applications, not just J2EE.

The goal of SABER is to be able to enforce these kinds of framework domain restrictions. In doing so, we can also exploit domain knowledge to perform a more efficient and precise analysis. For example, our race detection algorithm [28] makes use of the knowledge that certain *Servlet* methods are multi-threaded, and that synchronization is rarely used. This allows us to use an efficient algorithm which flags all updates to *Servlet* static and member fields as errors. While this algorithm would result in too many false positives in a general context, it performs well with a low false positive rate in the context of J2EE. Similarly, our filtering strategy exploits domain knowledge, so that we are able to reduce the number of false positives. We may know that a method inside a J2EE framework that appears to be callable based on the invocation graph, in fact, will never be called in a J2EE application. For example, one of the coding patterns detected by SABER is improper calls to *java.awt* methods from within a server side application, because such calls are not thread safe. Since the J2EE method *DatabaseConnection.connect()* calls *java.awt* methods, any application which includes *DriverManager.getConnection()* would normally result in an error report. However, we know from understanding the framework that these *awt* methods will never be called from a server application. Therefore, we filter out the reporting of this pattern.

3. RULES CATEGORIZATION

The 400 coding patterns that SABER detects are currently classified into six broad categories, based on the similarities in the algorithms used to detect these patterns. We refer to each such category as a *rule*. Each rule has one or more parameters. A specific coding pattern is derived from a rule by instantiating the rule, by providing specific values for each parameter. The rules themselves are general, with the intention that they are applicable to other frameworks besides J2EE. By creating a set of instantiated coding patterns, we customize the rules for a specific framework. Depending on the

rule, a parameter value can name a single artifact (e.g. class, field, method), or a set of artifacts, which can be described by attributes (e.g. all classes in a package, all static fields). This provides flexibility and easy rule management. Following are the SABER rules currently implemented

Do not call method X from method Y directly or indirectly. Much of the domain knowledge in Section 2 can be described by coding patterns instantiated from this rule. Here are a few examples:

```
Do not call java.lang.Thread.start()from javax.Servlet.     (violates 2.3)
Do not call java.util.jar from javax.Servlet.               (violates 2.6)
Do not call java.lang.Runtime.gc()from javax.Servlet.       (violates 2.6)
Do not call java.io from finalize().                        (violates 2.8)
```

As these examples illustrate, X and Y can be a package (e.g. *java.io*), a class (e.g. *java.lang.Servlet*) or a method (e.g. *finalize()*). If Y is a class, then the rule applies to all methods of all classes derived from the class Y (e.g. all classes derived from *Servlet*).

Must call X when Y. If ever method Y is called, method X must be called somewhere in the application. Note that for this category, there is no additional constraint that a path must exist in the application between Y and X. This rule is used for the problem described in 2.4:

```
Must call invalidate() when new HttpSession().
```

Note that since an *HttpSession* may be created in one web request, and invalidated in another request, there may in fact be no control flow path detectable between the creation and invalidation point.

Must call X after Y. If method Y is called at program point p, method X must be called on every path from p to the program end. In addition, the return or receiver object of method Y should be the same as the receiver of method X. This rule covers the defective coding patterns described in 2.7. For example:

```
Must call java.sql.Connection.close() after javax.sql.DataSource.getConnection().
Must call java.net.Socket.close() after new java.net.Socket().
```

Do not store objects of type X in Y directly or indirectly. This category is used for coding patterns enforcing domain constraints described in 2.1 and 2.5:

```
Do not store any object into a Servlet instance field.
Do not store any object into a static field.
Do not store any object that does not implement the Serializable interface
                                                        into HttpSession fields.
```

Do not extend X. Coding patterns in this category include instantiations of X where X is an interface or a class. For example from Section 2.1:

```
Do not extend SingleThreadModel.
```

Implement method pairs X and Y consistently. Coding patterns in this category include the constraint in 2.11:

```
Implement methods equals() and hashCode() consistently.
```

Section 6 describes our approach to program analysis of SABER rules, where each rule is implemented by a different analysis algorithm. Introducing a new rule requires supplying an additional algorithm to the system. The set of rules are by no means complete. Since SABER was based on our experience with finding problems in production code, the rules that were included in SABER were the ones that we found most useful in describing high-impact production problems with J2EE. For example, we include **Must call X after Y** to detect resources which are opened but not closed; however, we have not yet included **Must call X before Y**, which would detect resources which are closed but never opened. While also an error, closing a file without opening it is the type of error that is usually caught very early during testing, and does not survive into production. Future experience with SABER in other domains will result in extending the rule set.

4. FILTERING

Within each analysis for a SABER rule, false positive filtering is an important step. We define false positives broadly as an error that is reported by the tool, which is not really a problem from the standpoint of the overall application. False positives can be grouped into three categories depending on how they can arise.

4.1 Rule Specification Issues

If multiple rules can cover the same coding pattern, or different parts of a larger pattern, care must be taken so as not to introduce redundancy in the error reports. For example, *System.gc()* and *Runtime.gc()* are method calls that are forbidden. However, *Runtime.gc()* should only be reported when it is not called from *System.gc()*. This constraint is applied by external specification of the coding pattern (e.g., the pattern for the *Runtime.gc()* call includes *System.gc()* as a filter method). Such an external specification allows the user to customize the filters that are applied to each pattern. For example, this pattern is specified with a filter as:

```
Do not call java.lang.Runtime.gc()  from javax.Servlet,
                                        exclude method java.lang.System.gc()
```

When applying this pattern during analysis, the method *java.lang.System.gc()* and any code reachable from this method in the invocation graph is excluded from the analysis.

4.2 Domain Issues

Knowledge about frameworks used by the application can be used to exclude error reports when it is known that they cannot occur at runtime. This constraint is also applied by external specification of the coding pattern (e.g., the pattern which looks for calls to *awt* methods specifies *DatabaseConnection.connect()* as a filter method). This pattern is specified as:

```
Do not call java.awt from javax.Servlet,
                        exclude method com.ibm.db.DatabaseConnection.connect
```

As another example, the following pattern illustrates a filter which is a field:

```
Do not store any object into a Servlet instance field,
                                 exclude field java.util.Locale.defaultLocale
```

Since *defaultLocale* is initialized once at startup, we want to suppress reporting all stores into this field.

The domain-specific filters for each of the coding patterns implemented in SABER were derived from a single empirical study of a large application. Without any domain-specific filters, we had over 11,000 error reports from SABER on this application. After adding the filters, we reduced the reports to 30 odd reports, with no false positives in this specific application. We have since validated this filtering mechanism of several other commercial applications, and the results have completely generalized across these applications. In general, this filtering mechanism has reduced the number of false positive reports to more tractable numbers (0-22% of total number of reported defects). Section 6 describes how false positive filtering mechanism is implemented for different rules.

4.3 Application Issues

False positives may arise since static analysis cannot infer application usage models that are not explicit in the application. For example, *a Servlet* may invoke methods such as *Thread.sleep()* or *System.gc()*, coding patterns that can cause performance problems during program execution. However, the *Servlet* may only be used for administration purposes and hence only by one or two users at a time. In this application specific context, the *Thread.sleep()* and *System.gc()* are not problematic coding patterns. No program analysis tool can automatically filter out such false positives,

but the tool provides sufficient program context and explanations to help the user easily classify such reports as a false positives.

5. SABER ARCHITECTURE

Figure 2 shows the architecture of SABER: the input to SABER is Java object code, that is, classes that are stored in class files or J2EE EAR or WAR files. SABER also preprocesses Java Server Pages (JSPs) that constitute a good portion of most J2EE application code, and includes them within the application scope.

The basic analysis engine, JaBA[22][23], performs control and data flow analysis on Java object code and captures related information about classes and fields. The results of JaBA are used by the various SABER rule analysis algorithms that are built on top of JaBA. The SABER rules are encoded in a rules database, currently represented in XML format. This XML format allows the specification of parameters (X or Y) to the SABER rule, and includes a specification of the filter fields, methods, classes or packages that should be applied for a given SABER rule. The results of the rules analyses are fed to a framework that reports them in various formats: HTML or XML (when SABER runs in a batch mode) or within the context of in an interactive development environment (when it runs within an IDE).

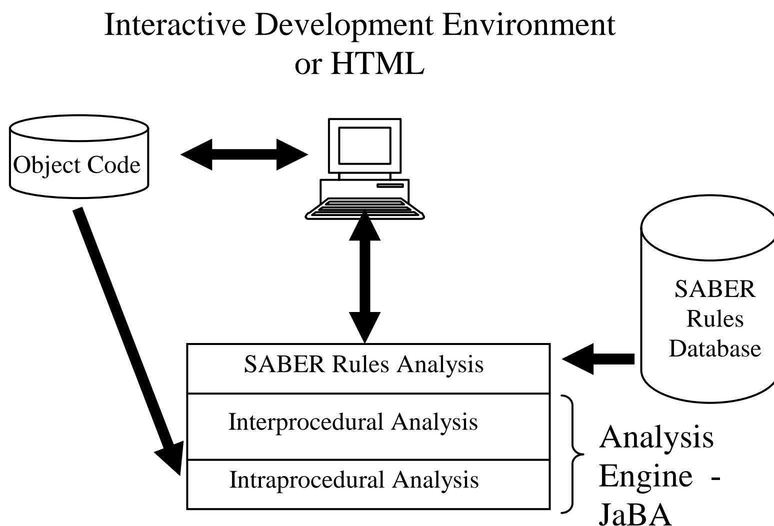

Figure 2: SABER Architecture

Analyzing an application involves the following steps:

Definition of the analysis scope. The analysis scope identifies the components of the application, framework and platform which are to be included as part of the analysis. For SABER this includes the classes and classloader tree for the various components used by the application. The user is given some control on the set of framework and dependent jar files they can include in the scope of analysis.

Analysis of application attributes. This provides basic application information such as the class hierarchy, the classes, fields, methods. At this step, JaBA generates the Class Hierarchy Graph data structure. The analysis to detect the SABER rule **Do not extend X** is performed at this step.

Perform intraprocedural analysis. This will identify basic blocks and build a Control Flow Graph [1] for each method and def-use chains [1] for reference variables used within the method. Analysis to check for the SABER rule, **Implement method pairs X and Y consistently** occurs in this step.

Select root methods. Root method selection identifies the entry points into the application. In the case of SABER and J2EE, root methods are the JSP, *Servlet* or EJB methods. This step is important because it limits the scope of the static analysis to methods that are relevant for the detection of SABER rules (*Servlet*/EJB entry points). In addition, because our rule detection algorithms only look at application code reachable from these root methods, this step also helps in improving the scalability of the analysis.

Perform inter-procedural analysis. The inter-procedural analysis includes global pointer analysis, generating an Invocation Graph [11] representing method calls in the application and a points-to graph (referred to as the connection graph in [8]) representing object and field references in the application. A majority of SABER rules are performed at this step.

Static analysis provides approximate results, which is a source of false positives. To provide a more accurate analysis, JaBA uses context sensitivity to distinguish between different potential uses of a given static construct. It is possible for SABER to miss bad coding patterns that exist in an application, that is, the SABER analysis is not sound. This is because of the various techniques, including filtering, that we use to avoid false positives. We have found in practice that there is a trade-off between a low false positive rate and soundness, which has forced us to engineer an acceptable solution, favoring false positive reduction.

6. SABER RULE ANALYSES

The SABER architecture framework allows for an arbitrary number of rule analysis plugins, providing rule extensibility. Each rule analysis component extends the *AnalysisObserver* class, and must have a corresponding XML descriptor. The descriptor indicates at which processing phase (see Section 5) the rule should be applied. We briefly describe the analysis algorithms for the non-trivial SABER rules described in Section 3.

6.1 Don't call X from Y

A method X must never be invoked on any call path that originates from method Y. This rule requires interprocedural analysis and relies on the invocation graph. The algorithm is a graph reachability problem. On any path in the invocation graph rooted at *Servlet* or EJB root methods, it checks for calls to X and Y. If both X and Y appear, it checks if X is reachable from Y.

For this SABER rule, progressively more expensive false positive filtering mechanisms are applied as follows:

1. If a call to X occurs on a path from a Y method, first check if any filter methods specified for this pattern are reachable from Y. Note that in this step, SABER uses the already cached result of reachable method calls from root nodes (Ys). Such a result is used across multiple rule analysis algorithms in SABER. If no filter methods are called, this X call should be reported as an error.
2. Check if the immediate callers of the X method call all are filter methods. If all immediate callers are filter methods, ignore this error report; otherwise, proceed to Step 3.
3. Remove all the filter method nodes from the set of reachable nodes from the root. If a path still exists from root to a "bad" method call, report the error.

6.2 Must call X after Y

This analysis, which is described in detail in [26], uses the invocation graph and the control flow graph. The invocation graph represents the method call information, containing edges from call sites to possible methods that can be invoked at that call site. The control flow information within each method is represented by the method's control flow graph where nodes represent a sequence of instructions that do not have a branch, and edges represent precedence information among these instruction sequences. Each control flow graph has a unique start node and one or more end nodes (corresponding to return statements in the method).

For each call site that invokes method Y (cy), the analysis searches for call sites, cx, that invoke method X on all paths from cy to the exit points of program. While doing this search, the algorithm looks for only those call sites cx whose receivers match receivers (or return types) at cy. The program exit points are essentially the end nodes of the root methods. When a call site other than cx or cy is encountered, the analysis traverses the control flow graphs of all the methods that can be invoked at that call site. The analysis also uses memoization techniques and invocation graph pruning techniques to optimize the path traversals. Memoization eliminates visiting control flow paths already visited and invocation graph pruning eliminates traversing methods that will not be on any path from cy to cx. A case that we have to filter out in this analysis is when the method X is enclosed in the following conditional code:

```
try {
 DataSource ds = InitialContext.lookup("jdbc");
 Connection conn = ds.getConnection();
 //some lines of code
} finally {
  if (conn != null)
    try {
      conn.close();
    } catch (SQLException e) {// ignore }
}
```

Here, Y is method *getConnection()* and X is method *close()*. If the *getConnection()* call, succeeds, the return type, `conn`, cannot be null. Hence, the path when *(conn == null)* can not be taken if *getConnection()* succeeds. The false positive filtering mechanism, integrated into the analysis, will ignore paths in the program when `conn` evaluates to null.

6.3 Do not store objects of type X in Y

This analysis [28] uses the points-to graph [11] whose nodes represent objects or fields of objects. Edges in the points-to graph either (a) represent an object stored in a specific field or (b) represents a field of an object. The analysis computes the closure of the points-to graph starting at "interesting" root nodes and accumulates all the objects and fields that are transitively reachable from the root nodes. While computing the closure, the algorithm only looks at objects and fields that satisfy a certain property (type X), and filters out fields based on the filter specification. For example, one "interesting" root node is the *HttpSession* object's field in which session attributes are stored. The corresponding property to check is if all objects that are stored in that field implement the *Serializable* interface. Using this root node and property, we can detect if a non-serializable object is stored directly or indirectly in *HttpSession* objects. An essential case

that we filter out is when fields within the *Serializable* object are declared transient, which occurs while computing closure of the root node.

6.6 Implement method pairs X and Y consistently

In general, determining whether two methods are implemented consistently is a difficult problem. The SABER analysis for this rule is simple, efficient, and approximate. We compare the member fields accessed in each of the methods X and Y. If the accessed fields are different, then there is a potential inconsistency. For example, if a write method accesses fields A, B, C, and the related read method accesses only A and B, then it is likely that what is read is not the same as what is written, and the method pair implementations violate an implicit contract.

7. CUSTOMIZATION OF PROGRAM ANALYSIS FOR JAVA/J2EE SEMANTICS

The semantics and common usage patterns in J2EE require customized static analysis processing. In some cases, customized processing is necessary to make the analysis more scalable or precise. We describe several examples of J2EE analysis customization.

7.1 Reflection

Even though it is bad practice to use reflection in J2EE applications for performance reasons, reflection occurs frequently, for example, when the transaction input includes a string to specify a command method. Therefore, it is important for SABER to be able to effectively analyze code with reflection. Dynamic call paths occur when users use the reflection API to dynamically generate classes through the *Class.forName(String)* method, or dynamically invoke methods on these classes through the *Method.invoke(String, Object[] args)* methods. From a program analysis perspective, these calls are problematic because it is often unclear which class/method is being invoked. String propagation to determine the class/method is often not useful, because for example, the strings used to instantiate classes or invoke methods are often read from properties files. Therefore, in SABER, we adopted the approach of observing the casts from specialized methods (e.g., *Class.forName()*) to determine the class to be instantiated. In our experience with commercial applications, it is frequently the case that the result of the cast from a *Class.forName()* method is frequently an interface. In this case, all possible concrete classes that implement the interface are included as potential callees, because this approach is deemed better in terms of usability from our field engineers. In our tests of the tool with commercial applications, we have observed cases where this assumption appears to be justified. Specifically, a common pattern in Web applications is the

delegation of the processing of a *Servlet* request to different handlers based on the parameters of the request; which means that any of the different concrete implementations of the interface can indeed be instantiated at the point of the *Class.forName()* call. Note that the approach of using casts to infer a dynamic class fails to handle the case where methods are invoked dynamically; this is a limitation of SABER.

7.2 Struts

Struts [29] is a commonly used Model-View-Controller framework that is built on top of *Servlets.* In struts, a single *ActionServlet* processes all requests, and delegates to different methods of *Action* objects based on an external XML based mapping of requests to methods. Because the *ActionServlet* uses reflection to dynamically invoke methods, we cannot use the mechanisms to handle reflection to handle struts-based applications. Instead, SABER automatically detects if the application is a struts application, and includes all *Action* objects and their methods as root methods within the call graph.

7.3 Web Services

Web services is another framework that depends on external XML based mapping of requests to arbitrary Java objects and methods. In this framework as well, methods are invoked using reflection by a single *Servlet*, with the difference being that any method can be invoked. To handle this case, SABER determines if the application is a web services based application, parses the XML based mapping of requests to Java objects/methods, and includes these methods as roots in the call graph.

7.4 EJBs (Enterprise Java Beans)

In the EJB framework [13], there are two important players: the application code that customizes the EJB interfaces to lookup, create, and invoke business methods on an EJB; and the *container* code that provides implementations of these interfaces which are composed of both EJB and application-specific semantics. EJB semantics handle functions that are common to all business methods, such as persistence, transaction-handling, security, access control, etc. Application-specific semantics are business methods that are unique to a given application, and an implementation of these methods is provided by the client code. The relationship between the container code and the application code is as follows. The application code customizes the *EJB Home Interface*, the *EJB Component Interface* and the *Enterprise Java Bean class,* which are each extensions to interfaces and classes respectively defined in the EJB

specification. A complex relationship exists between the *EJB home interface*, the EJB *component interface*, and the *EJB bean class*. *EJB home interface* specifies the methods to find or create EJB components. The application code can customize the interfaces that the application can use to create or find EJB components, but the container provides the concrete implementation of the EJB Home object, so that it can manage the lifecycle of the EJB components that are created within an application. The EJB component interface specifies the business methods of an EJB component. Again, the application code can customize these interfaces to provide methods unique to the application, but the container provides the concrete implementation of the EJB component interface to a client, so that it provide EJB semantics such as persistence, transaction handling, security, etc. when a client invokes business methods on an EJB component. The EJB bean class allows the application object to provide the implementation of the finder methods and business methods specified in the customized EJB Home and EJB Component interfaces, respectively Under the covers, for every customized method call whether it is a business method or a finder method, the container generated implementations of the EJB Home interfaces and the EJB Component interfaces will invoke the corresponding methods defined in the application's bean implementation class (*EJB Bean Class).* The relationship between the EJB Home, EJB component, and *EJB Bean classes* is specified in an external XML deployment descriptor file.

From a program analysis perspective, there are two choices in terms of dealing with the EJB model: (a) to use the container generated classes as return values for a call to look up an *EJB Home*, or calls that create *EJB objects*, or (b) to synthesize classes from the application code that satisfy the type hierarchy required from a program analysis perspective. We chose the latter option, because it greatly reduces the complexity of the invocation graph [15]. Specifically, we create a synthesized container that essentially reads the deployment descriptor for the application code and generates a synthesized *EJB Home* object and a synthesized *EJB Object* for each EJB type. When the client invokes a JNDI lookup, the return is of declared type *EJB Home Interface,* but the concrete type is ``synthesized *EJB Home class''*. For each EJB specified in the deployment descriptor, our static analysis engine generates this synthesized *EJB Home class.* The class contains methods corresponding to all methods declared and inherited by *EJB Home Interface.* The create methods in this class return objects whose declared type is *EJB Component interface,* but the concrete type is a ``synthesized *EJB Component Class''*. Our static analysis engine generates the *EJB Component class,* including methods in this class. This class implements the *EJB Component Interface* and provides implementations for all methods declared and inherited by this interface. For all those methods, such as business methods, that have implementations in the EJB's bean class (defined

by the application), the *EJB Component class's* synthesized methods are essentially wrappers to the corresponding methods defined in the EJB bean class.

7.5 Container pattern

SABER provides *method sensitivity* to distinguish between different calls to the same method, and *class sensitivity* to distinguish between different allocations of the same class. The *method sensitivity* used in SABER is a modified version of the Cartesian Product Algorithm [1], where the sets of types of each argument are used to determine what context of the target method to use. The *class sensitivity* most often used in SABER is *allocation-based sensitivity* where instances are differentiated based on the byte code offset within a method where the allocation occurred. Early experiments with SABER suggested that we needed to vary class sensitivity for the **Do not store X in Y** analysis to accommodate the *container pattern* often found in the Java collection classes, but also in some non-collection classes (a detailed discussion of this issue can be found in [28]). As shown in Figure 3 below, *FilterInputStream* contains an instance field *in*, which is assigned the value of a wrapped stream at the time the constructor is called. A call to close the stream gets delegated to a call to close the internally wrapped stream *in*. If multiple calls exist to the *FilterInputStream* constructor, and only one of the streams gets closed, method based sensitivity in this example will incorrectly suggest that all streams were closed.

```
public class FilterInputStream extends InputStream {
    /**
     * The input stream to be filtered.
     */
    protected InputStream in;

    protected FilterInputStream(InputStream in) {
       this.in = in;
    }

   public void close() throws IOException {
       in.close();
    }
}
```

Figure 3. FilteredInputStream.java

For better precision, we need *receiver-based sensitivity*, where allocation sites are augmented with the allocation site of the receiver of the method where the allocation occurs. For scalability reasons, *receiver-based sensitivity* is

selectively used for classes that we identified in our early experiments as being problematic either because they produce misses, or false positives. These classes are largely the container classes in Java (e.g., Vector, Hashtable, etc.), and the stream classes as illustrated in the Figure 3.

8. PROGRAM DEFECT UNDERSTANDING

The types of analyses outlined in Section 8 identify defective coding patterns that often span multiple classes, methods, and object instances. Whether these patterns represent a real defect or a false positive from the standpoint of the overall application is often not readily apparent to the user. It is also not clear to the user what remedial actions they need to take to eliminate the defect. To enhance program defect understanding, the tool provides two broad categories of supporting information: (a) an explanation of why the code pattern is defective, along with a set of mitigating circumstances that might occur in an application which would make the code pattern a false positive, and (b) selective display of contextual path and data flow information which can explain how the defect occurred. We discuss each in detail below.

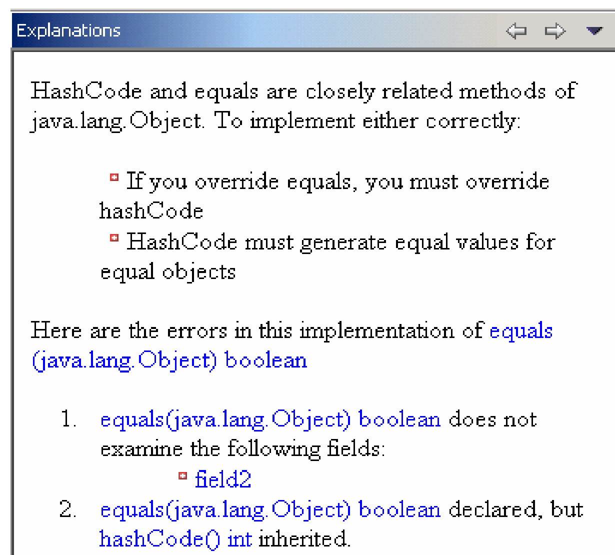

Figure 4: Explanation view in SABER

8.1 Explanation of the Defect

For a certain class of defects, such as **Do not store objects of type X in Y**, a precise explanation of why a certain method is incorrectly implemented is sufficient for defect understanding, and corrective action. Figure 4 shows an example

from the tool on a sample program where the coding error is a violation of the contract between the *hashCode()* and the *equals()* method. In this specific example, the tool lists two of the five possible reasons from the equals-hashCode contract why this is a violation. The first reason is less serious, and falls in the category of a warning; one of the fields (*field2*) of the object is not examined in the *equals()* implementation. The second reason is more serious. The *equals()* method of the object has been overridden, but the *hashCode()* method is inherited, which means that the contract between *equals()* and *hashCode()* has been violated.

The explanation view shown in Figure 4 is completely integrated in the IDE, such that clicking on the method or field names shown in the example leads the user to the corresponding methods or fields in source, thus enabling easy navigation. This view is also useful for explaining mitigating circumstances where a certain coding defect would not pose a problem. For instance, if a *Servlet* extends *SingleThreadModel*, the tool classifies it as a defect, because it has negative implications for performance. However, if the *Servlet* is only accessed by a small number of people (e.g., for administration tasks), then the coding pattern is not problematic. A description of these circumstances can help the user decide if this is a serious problem.

8.2 Presentation of Program Context

For the types of defects reported by SABER, an important aspect of program defect understanding requires an understanding of the context within which an error occurred. This typically involves the presentation of control flow information, or the presentation of both control flow and data flow, based on the type of defect.

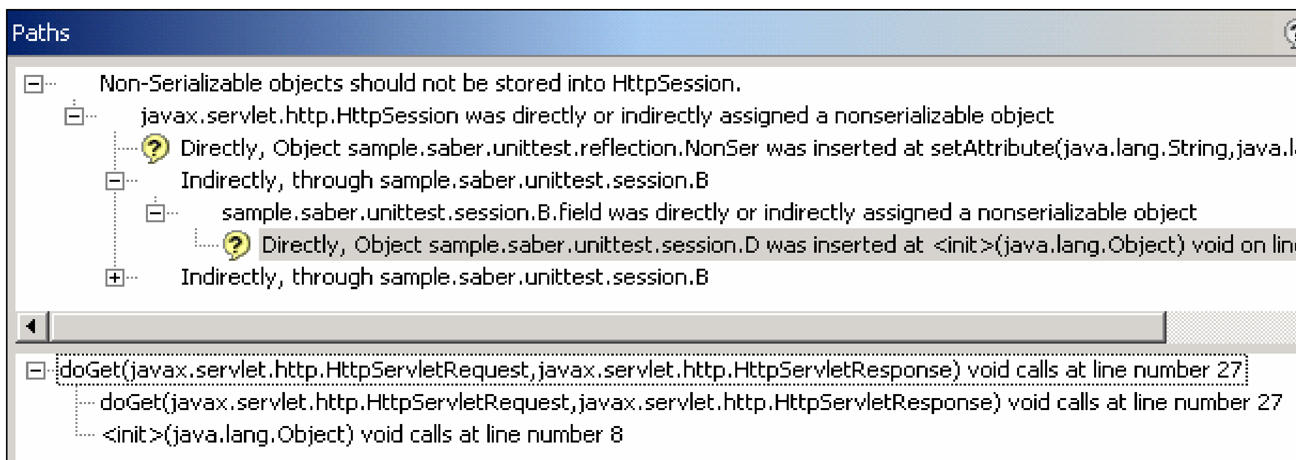

Figure 5: Data flow information in SABER messages

Control Flow presentation: Defects that are path related (e.g., **Do not call X from Y**, or **Must call X after Y**) require a display of paths through program code for an understanding of the defect. A key issue in the display of control flow information is how to limit and select from the large number of paths in the program to the erroneous code. Our heuristic for path selection is to choose paths based on two considerations. The first consideration is that we need to provide information that is important for the identification of false positives. Our observation is that frequently, root methods can provide information that is critical for this determination (e.g., if an administrative servlet makes calls that will have an adverse effect on performance, it is not a defect from the standpoint of the overall application). The second consideration is to show information important for remedial action. Our observation here is that frequently, the program point that needs modification is the caller method to the X in question. To satisfy these two considerations, we provide (a) at least one path for each unique Y method from which X is called (b) at least one path through all unique call sites from which X is called. This heuristic allows us to limit the number of paths to 10 or less paths, rather than overwhelm the user with hundreds of possible paths in the program. Example SABER rules that use this form of path related displays include **Do not call X from Y**, **Inconsistent method implementations** (where path related information can determine if the incorrect method implementation is reachable from roots), and **Must call X from Y**.

A second form of path selection is required for the SABER rule **Must Call X after Y**, e.g., closing of database connections after creating them. Here, selected paths show at least one path from each unique Y method to the return from the Y method where there are no X calls in the path.

Data Flow presentation: Defects related to **Do not store objects of type X in Y** typically require the presentation of both data flow and control flow information for defect understanding. Take as an example, the defect of stores of non-serializable objects into the *HttpSession* object, as shown in Figure 5. In this case, object D is non-serializable, and it gets stored indirectly into *HttpSession*, through object B. The top panel displays a tree view of direct and indirect stores of non-serializable objects into *HttpSession*. In the case of the indirect stores, each insertion of the contained object into the container object is shown and hyperlinked to source. This helps the user understand how object D got inserted into *HttpSession*. The lower panel contains the corresponding control flow information; i.e., path(s) from one or more unique Y root methods to the insertion point appears in the lower panel when the user clicks on the insertion point. Thus, while the top panel shows the containment hierarchy of how objects were stored into a container of interest, the lower panel provides reachability information about how the defective insertion point could in fact be reached from a root method. The two

together again provide information useful for remedial action, or for ruling out a defect as a real problem because of mitigating circumstances.

9. VALIDATION

To perform the SABER rule analysis, analyzing the application alone is not sufficient because object allocations occur in and control flow paths pass through other code that the application depends on. This other code includes the J2EE framework, any third party libraries, and the JDK itself, all of which must be included in the analysis. However, while all of the additional code must be included in the analysis scope, only code which is reachable from one of the root methods will be included in call graph construction. The outlined area in Figure 6 represents the reachable portions of the application and framework code.

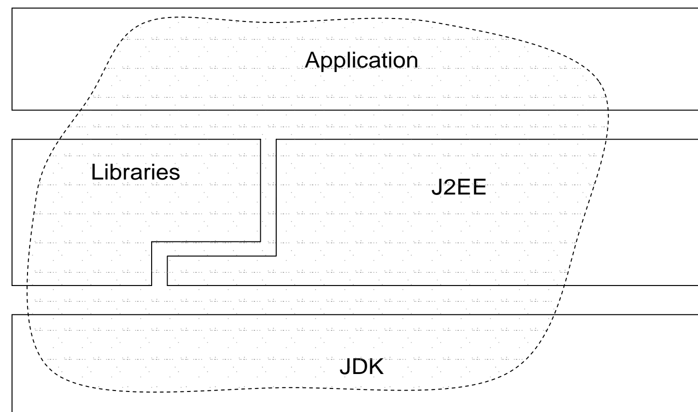

Figure 6. Application Reachable Portions

SABER has been validated on several commercial applications, labeled A1-A4 in Table 1. A1 is an application for ordering products in a large appliance company. A2 is an application from a large technology company that tracks skill development and placement of employees within the company. A3 is an order tracking application for products in a large technology company. A4 is a financing application, i.e., manages payments, credit applications etc. Table 1 shows the breakdown of classes analyzed by SABER for each of these four commercial applications.

	A1	A2	A3	A4
App. Classes	211	974	135	730
App. Classes Analyzed	208	730	48	694
Synthesized	6	15	4	33
Framework Classes	2847	1767	855	869
Frameworks Classes Analyzed	1296	87	31	66
JDK Classes	5712	6432	5722	5620
JDK Classes Analyzed	429	437	403	419
Total Classes	8770	9188	6716	7252
Total Classes Analyzed	1933	1254	482	1179

Table 1: Analysis breakdown for applications

Table 2 gives the total number and averages per method of control flow edges and nodes and average fanout at control flow branch points for three of the applications (A2-A4). The numbers are taken on code that is potentially reachable from J2EE starting points such as *Servlet* methods, and gives some idea of the scalability of the analysis.

Application	#methods	#basicBlocks	#Edges	#avg BasicBlocks	#avg Edges	#avg Fanout
A2	11,577	239,794	575,558	20	49	2.4
A3	2,032	36,741	65,358	18	32	1.78
A4	5,996	92,195	174,445	15	29	1.89

Table 2. Control Flow Complexity

Table 3 shows the results for the four commercial applications we analyzed. All of the SABER runs were on a 1.2 GHz, 1G memory laptop. The second column in the table gives the time it took for SABER to analyze the applications: this includes time to preprocess WAR or EAR files, perform intra and interprocedural program analysis and execute the analyses of all the SABER rules discussed in Section 2. The third column gives the number of messages generated by SABER including the number of false positives. We determined the false positives by inspection; it wasn't a difficult task since we had enough context information, i.e., control and data flow paths, for each message.

App	Time (mins)	#Messages/False Positives
A1	12.11	3 (0)
A2	8.6	49 (11)
A3	1.13	14 (2)
A4	16.83	13 (2)

Table 3: Saber time and number of messages

Table 4 shows the breakdown for each SABER rule analysis currently implemented in SABER on 11 customer applications B1-B11. (Applications A2-A4 are included in this collection.) The last column indicates the total number of false positives for each analysis; the last row shows the total number of false positives for each application. These numbers include cases that were classified as false positives based on application specific information (e.g., storing objects into a static *Hashtable* indexed by thread ID is thread safe because of the nature of the key, but this store would be flagged by SABER as an instance of writes to shared state). The most common rule violated is the **Don't Store X in Y rule**. Most of these violations are race conditions (193 out of 266). This rule also has the highest false positive rate (19%). In total, SABER found 455 bad coding patterns and 79 false positives in these 11 applications, which is an overall false positive rate of 15%.

Applications/ Saber Rule	**B1**	**B2**	**B3 (A3)**	**B4 (A4)**	**B5**	**B6**	**B7**	**B8 (A2)**	**B9**	**B10**	**B11**	**Total**	**Total False Positives**
Implement Method Pairs consistently	0	10	0	1	4	4	0	13	102	0	2	**136**	**6**
Don't Extend X	0	0	0	0	0	0	1	0	0	0	0	**1**	**0**
Don't Store X in Y	33	61	12	0	17	2	48	16	0	48	29	**266**	**61**
Don't Call X from Y	0	5	1	10	0	0	0	2	0	5	0	**23**	**1**
Must Call X	0`	1	0	1	0	1	1	0	1	0	0	**5**	**0**
Must Call X after Y	3	4	1	1	2	0	0	7	0	0	6	**24**	**3**
Total	**36**	**81**	**14**	**13**	**23**	**7**	**50**	**38**	**103**	**53**	**37**	**455**	
Total False Positives	**6**	**17**	**0**	**0**	**4**	**0**	**12**	**11**	**4**	**12**	**5**		**79**

Table 4: Breakdown for each SABER analysis

10. EXTENSIBILITY

The ability to extend or customize the coding patterns that a program analysis tool detects is important for the following reasons. First, users will invariably want to check for patterns which are different from what the tool provides. This is because development groups may often have their own coding conventions and rules they wish to enforce for their custom frameworks. Second, additional components are always being added to any platform. For example, new components are continually being added to J2EE and the rule set must be easily extensible to accommodate these new components. In SABER, extensibility has taken the form of allowing users to change the parameters to some of the SABER rules. This allows a user to enforce project wide policies.

Using this mechanism, we have extended SABER to areas beyond the J2EE framework with success. Although the rule categories were built by studying common J2EE errors and best practice violations, it is notable that verification analyses from other areas have mapped well into these categories. This shows that a problem categorization approach like the one in SABER has applicability beyond purely J2EE analysis.

One of the areas outside of J2EE in which SABER was applied was National Language Support (NLS) and Internationalization Compliance Testing (ICT). From examining NLS and ICT validation tools as well as working with a NLS ICT validation group, we found 19 different problematic coding patterns in this domain. Of these 19 patterns, 15 fit into the "Do not call X from Y" category of SABER. The remaining 4 coding patterns dealt with String constant and character usage which do not fit into a current SABER rule category. The coding patterns which fit into SABER rule categories included:

- Do not call methods: e.g., *String.concat(), String.equals(), String.compareTo()*.
- Do not call *toString()* on the following classes: e.g., *Date, Time, Number, BigDecimal, BigInteger*.

An interesting finding of NLS and ICT verification with SABER on several commercial applications was that we found that many components presumed to be safe from an NLS and ICT standpoint were actually not safe. This was because many J2EE extension components internally called methods which were unsafe for NLS and Internationalization. This finding led to the addition of approximately 27 more coding patterns to the existing coding patterns listed earlier in this section. In addition to applying SABER to NLS and ICT, SABER has been applicable in several other areas. These include code conversion such as migration of an application to a newer set of framework APIs and porting such as identifying platform specific component usage when moving an application from one J2EE container provider to another.

11. RELATED WORK

Program analysis tools and specialized checkers such as Lint[21], ITS4 [28], JTest [25] and FindBugs [17] provide useful information about code violations, but they are limited in the types of violations they can detect to mostly errors that do not require detailed program analysis. Annotation checkers such as LCLint [14], Aspect [18] and Extended Static Checker [10] employ program verification techniques to identify defects. Unlike SABER, these tools enlist the programmer's assistance in adding annotations to the code that can be used to verify the correctness of the program. Other software model checking tools such as SLAM [5], MOPS [7] and Bandera [9] and static analysis tools such as xgcc [12][16], have proven to

be useful in detecting bugs in some large C programs. Several of the Saber rule categories (eg Must Call X after Y and Don't Call X from Y) can be coded as Metal rules and verified in xgcc. Metal is a more general rules language, and rules can be encoded in Metal that are not currently supported in SABER. However, rules such as "Do not store objects of type X in Y" cannot be coded in Metal, since this rule requires analysis of the object reference graph. The ESP tool [4] uses a combination of *scalable* path-sensitive static analysis and property simulation, a new method of partial verification to detect errors in large C/C++ programs. ESP does not analyze Java applications but is close in spirit to SABER. SABER differs from ESP in that it does not employ path-sensitive analysis and achieves scalability by combining domain knowledge with static analysis. From our experience with commercial J2EE applications, we have observed that common problematic coding patterns in these applications are not the same as in general Java or systems C/C++ programs. We also haven't seen the need for complex path-sensitive analysis in filtering the common false positives. PREfix [6] is a static analysis tool that detects defects in large C/C++ programs and libraries and automatically simulates execution of source code components without requiring test cases or instrumentation. SABER does not attempt to simulate program execution.

Unlike most of the above mentioned tools, an additional advantages in SABER are the categorization of rules into higher level abstractions, the domain-knowledge-specific filtering to avoid false positives, and the contextual reporting information, all of which assist tool usability. The checking of violations can be customized by simple changes to a rule database, i.e., adding new rules or modifying existing rules. These changes may be done due to platform evolution or if SABER is applied to other problem domains.

An abridged version of this paper has appeared in [27].

12. CONCLUSIONS

In this paper, we have presented SABER, an analysis tool, which automatically detects subtle, but serious defects in large commercial J2EE applications. SABER combines a body of domain-specific knowledge gathered from actual field studies with the results of static analysis to aid in the detection of such defects. SABER has the additional advantages of being relatively fast, extensible to other problem domains, and adaptable to changes in best practice patterns. SABER has already had successes in identifying defects in large commercial applications. In field studies, the types of defects detected by SABER have typically caused serious problems in production, from system outages to incorrect program behavior. The fact

that SABER can be used to detect these problems prior to deployment of code into production makes it a valuable tool for IT organizations.

ACKNOWLEDGMENTS

We thank Gary Sevitsky, Nick Mitchell, Vas Bala and the anonymous reviewers for comments on the paper. We also thank Chet Murthy for his valuable insights.

REFERENCES

[1] O. Agesen. The Cartesian product algorithm: simple and precise type inference of parametric polymorphism. *Proc ECOOP '95*, August 1995, Springer-Verlag 1995.

[2] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools*. AddisonWesley, Reading, MA, 1986.

[3] D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies,* Prentice Hall, June 2001.

[4] M. Das, S. Lerner, M. Seigle. ESP: Path-Sensitive Program Verification in Linear Time, In *Proc. of PLDI 2002*.

[5] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis, In *Proc. of the 29th POPL*, January 2002, 1-3.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30 (7), June 2000, 775-802.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software, In *Proc. of the Ninth ACM Conference on Computer and Communications Security* (Washington, DC, November 2002), 235-244.

[8] J-D Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, S. Midkiff, Escape Analysis For Java, In *Proc. of OOPSLA*, October 1999, 1-19.

[9] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code, In *Proc. of ICSE*, June 2000.

[10] D. L. Detlefs. An overview of the extended static checking system. *SIGSOFT Proceedings of the First Workshop on Formal Methods in Software Practice*, January 1996, 1-9.

[11] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers, In *Proc. of PLDI*, June 1994, 242-257.

[12] D. Engler, B. Chelf, A. Chou and S. Hallem, Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, In *Proc. of SOSP*, October 2000, 1-16.

[13] Enterprise JavaBeans™, Specification version 2.1. Sun Microsystems.

[14] D. Evans. Static Detection of Dynamic Memory Errors. In *Proc. of PLDI*, May 1996, 44-53.

[15] S. Fink, J. Dolby, L. Colby, Semi-automatic J2EE transaction configuration. IBM Research Report RC23326, 2004.

[16] S. Hallem, B. Chelf, Y. Xie and D. Engler, A System and Language for Building System-Specific, Static Analyses, In *Proc. of PLDI*, June 2002, 69-82.

[17] D. Hovemeyer and W. Pugh, Finding Bugs is Easy, http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf

[18] D. Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Trans. on Software Engineering and Methodology*, 4 (2), April 1995, 109-145.

[19] Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification, Sun Microsystems.

[20] Java™ 2 Platform, Enterprise Edition Specification, v 1.4 API Specification, Sun Microsystems. 11/24/2003.

[21] S.C. Johnson. Lint, a C program checker. Unix Programmer's Manual, 4.2 Berkeley Software Distribution Supplementary Docs; U.C. Berkeley, 1984.

[22] L. Koved, JABA-JAva Bytecode Analysis in http://www.research.ibm.com/javasec/JaBA.html.

[23] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java, In *Proc. of OOPSLA*, Nov 2002.

[24] Object Technology International, Inc. Eclipse platform technical overview, July 2001, http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[25] Parasoft Corporation. Automatic Java[TM] software and component testing: using Jtest to automate unit testing and coding standard enforcement, http://www.parasoft.com/jsp/products/article.jsp?articleId=839&product=Jtest.

[26] D. Reimer, A. Kershenbaum, E. Schonberg, K Srinivas, H Srinvasan. Static validation of resource management in large Java applications. *First Proc of ISOLA*, Cyprus, Oct. 2004.

[27] D Reimer, E. Schonberg, K Srinivas, H Srinivasan, B Alpern, RD Johnson, A. Kershenbaum, L Koved. SABER: Smart Analysis Based Error Reduction. *Proc. ISSTA*, July 2004.

[28] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, J. Dolby, A. Kershenbaum, L. Koved. Validating structural properties of nested objects, *Proc. of OOPSLA 2004*, Vancouver, CA, Oct 2004.

[29] Struts, Apache, http://struts.apache.org.

[30] J. Viega, J. T. Bloch, T. Kohno, G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code, In Proc. of the 16th Annual Computer Security Applications Conference, December 2000, 257-269.