Towards Life-long Software Self-validation in Production

Daohan Qu
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
dawn@smail.nju.edu.cn

Chaoyi Zhao
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
cyzhao@smail.nju.edu.cn

Yanyan Jiang
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
changxu@nju.edu.cn

ABSTRACT

The increasing complexity of software and its execution environment makes in-house software testing challenging. Field testing, which conducts software testing in production environments, is a potential solution to this issue. However, existing field testing systems have not seen widespread use due to their inconvenience, lack of generality, and limited capabilities. We identify four essential requirements that a practical field testing system must fulfill: robust, efficient, handy, and versatile. This paper presents the design and implementation of Jaft, a field testing system for Java software meeting the aforementioned requirements through its design of field testing API, isolation mechanism, and runtime module. Evaluation results show that it has acceptable runtime overhead and can improve test effectiveness.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

field testing, in-vivo testing, field failures

ACM Reference Format:
Daohan Qu, Chaoyi Zhao, Yanyan Jiang, and Chang Xu. 2024. Towards Life-long Software Self-validation in Production. In *15th Asia-Pacific Symposium on Internetware (Internetware 2024), July 24–26, 2024, Macau, Macao.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3671016.3671382

1 INTRODUCTION

Software validation is challenging because software in a production environment must operate under various software/hardware configurations, underlying platforms, user inputs, and other factors, usually different from those of in-house validation. A study of 119 software *field failures*–failures that occur in actual usage–from three well-maintained systems (Eclipse, OpenOffice, Nuxeo) reveals that 70% of these failures are extremely difficult to detect using traditional in-house testing methods [13].

One promising direction to tackle this challenge is making software validation both in vivo and production, also referred to as *field testing* [4, 13, 19]. The seminal work CrystalBall [25] takes distributed snapshots of distributed systems and exercises the follow-up state space via model checking, opening a new direction to validation in production. In-vivo testing [5, 6, 9, 10, 19] advocates revealing software faults of unanticipated execution states that appear in production by executing in-vivo test cases, similar to a unit test, within the running production software to uncover faults.

Despite that the idea of "testing in the field" is simple yet powerful, it has not been widely adopted because of the considerably high risk, cost, and human labor for deploying such intrusive mechanisms like checkpoints in production. For example, StealthTest [7] relies on transactional memory. CrystalBall restricts the SUT to be Mace-implemented [14]. In-vivo testing requires users to provide in-vivo tests, which are sometimes beyond the common knowledge of ordinary developers [19].

Problem Formulation. This paper identifies the obstacles impeding practical field testing, yielding the following critical requirements for a *usable* field testing system:

- **Robust**. Robustly support field testing of *any* application with arbitrary programming language features on a specific platform (in this paper, for example, OpenJDK).
- **Efficient**. Have configurable and controllable runtime overhead to meet the QoS requirement of the underlying production services.
- **Handy**. Do not need sophisticated configurations and specific expertise to integrate field testing with the development and release workflow.
- **Versatile**. Can integrate a broad spectrum of testing and validation techniques for hunting a wide range of bugs.

To the best of our knowledge, no existing field-testing technique simultaneously satisfies the above requirements.

Internetware 2024, July 24–26, 2024, Macau, Macao

ACM ISBN 979-8-4007-0705-6/24/07
https://doi.org/10.1145/3671016.3671382

Jaft. This paper presents the design and implementation of Jaft, a practical field testing framework for Java. The design principles are highlighted below:

- *Separation of mechanisms and policies.* We decompose the field-testing as front- and back-end, with a separate control plane for system configuration.
- *Minimized intrusion to the Java runtime.* We limit our modification to the underlying runtime (OpenJDK 11) minimal. The only side-effect is creating a `fork` snapshot. Defects in the rest of the system (e.g., validation implementations) will not impact the production-run behavior.
- *Customizable test space exploration.* We provide a lightweight mechanism for developers to specify testing strategies in place.

To integrate Jaft in an existing code base, developers first include Jaft's field testing library and use the choice API to suggest field testing start time and declare a *search space* consisting of the divergent paths that can be explored during field testing, then simply replace the `java` command with `jaft`, which behaves exactly like a standard Java Virtual Machine, to run the program and it is automatically enabled with field-testing capability. We also provide an auxiliary library for replaying a designated choice sequence for debugging the field-testing code.

At runtime, Jaft `forks` the entire process at developer-specific program points, parses the runtime data area of Java Virtual Machine to obtain a continuable program state snapshot, and sends it to the coordinator for validation, as shown in Figure 1. Snapshots are later loaded by a validation VM (specifically, Java Pathfinder) for automatic state-space exploration. We also provide a command-line tool for dynamically configuring the parameters of the production and validation VMs.

Contributions. In summary, this paper makes the following contributions:

- We design and implement Jaft, a field testing system for supporting robust, efficient, handy, and versatile field testing.
- We conduct overhead and effectiveness experiments on Jaft, showing that Jaft can control its runtime overhead, and developers can leverage their knowledge to conduct effective field testing.

Organization. The rest of the paper is organized as follows. Section 2 describes the design of our field testing system, including field testing API, isolation mechanism, and runtime module. Section 3 introduces the system implementation, mainly about some technique challenges. Section 4 presents the experimental results and section 5 concludes the paper.

2 SUPPORTING LIFE-LONG SOFTWARE SELF-VALIDATION

We implement Jaft field testing system over Java Virtual Machine (JVM), a popular platform for enterprise applications. The workflow of Jaft is shown in Figure 1, which also illustrates the three main components of Jaft's runtime module: *production runtime (VM)*, *validation runtime (VM)* and a *coordinator*. We then discuss our design choices to meet the requirements for a usable field testing system:

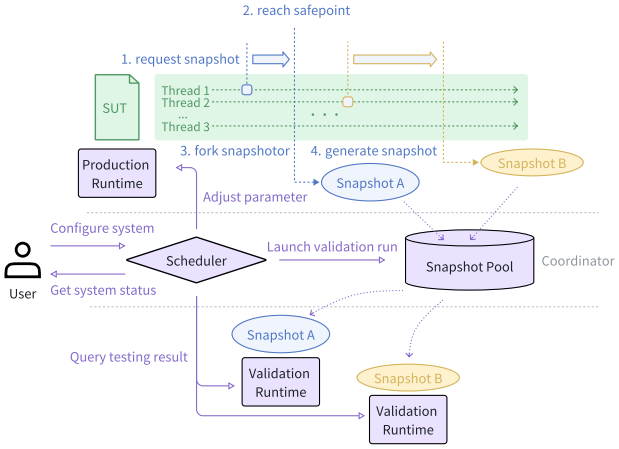


Figure 1: Field testing workflow of Jaft

Robust. The production runtime of Jaft is a patched version of OpenJDK 11. In pursuit of a robust production runtime, modification to OpenJDK has been minimized and its side effects have been confined: snapshot request reuses JVM's existing internal mechanism, generation of snapshot happens in a forked process, and validation of snapshot also executes in another process, the error in which won't affect the original running program.

Efficient. Our runtime module offers a command line interface for dynamic system configuration, allowing users to set CPU and memory load limits for field testing activities. The scheduler in our system's runtime module then uses these settings to efficiently manage snapshot validation.

Handy. We design a set of field testing APIs for users with some knowledge of the SUT to write field testing code thus guiding the testing process. Users could use the choice interface to suggest field testing start time and declare testing space, which is a search space regarding uncertainties in program execution. A decision interface is provided for users to easily customize testing strategies, which specifies how to traverse the testing space. We also provide a debugging interface for users to quickly check their field testing code before deployment. The above three types of interfaces constitute our field testing API and make the system handy to use.

Versatile. We design a Java program snapshot format and serialize the running Java program state into this format. This format could run in the validation runtime, where a broad spectrum of validation techniques could be applied. In such a manner, we bridge the program running in production to various existing testing techniques.

The rest of this section describes the design of Jaft field testing system.

2.1 Field Testing API

Testing is all about driving a program into a *state* that violates a specification by controlling nondeterminism in its execution. These nondeterministic factors constitute a search space, where testing tries to find a path to an erroneous state of the program. Jaft starts the search from a production snapshot, over the following mechanisms, which are listed in Table 1. Using

Table 1: Field Testing API

Interface	Category	Description
`void desireSnapshot(String name)`	Choice interface	suggest that a snapshot could be generated
`boolean isValidationRun()`		check if it is running in validation runtime
`String choice(String[] choices, String name)`		provide multiple choices for uncertainties
`List<ChoiceInfo> getCurrentChoices()`	Decision interface	get all available choices
`StateTransition proceedWithChoice(ChoiceInfo c, int traceLevel)`		set the value for a choice and proceed
`void pauseAt(DbgPauseReason reason, String name)`	Debugging Interface	set a pause point during debugging
`void noPauseAt(DbgPauseReason reason, String name)`		remove a pause point during debugging
`void proceedWith(String choice)`		set a choice and proceed during debugging

these APIs, the developer specifies points of potential snapshots and defines the search space over possible points of non-determinism.

2.1.1 Choice interface. The primary functions of the choice interface are listed in Table 1. It enables developers or testers to provide effective testing methods utilizing their domain knowledge. At each point facing uncertainties in the program, they could use the `choice()` function to set multiple choices for the uncertainties, which are more prone to reveal bugs. When the program tries to explore these different choices in the validation runtime, the testing space is traversed.

An example of using the choice interface to write field testing code is shown below:

```
 String userName = getUserName();
 String password = getPassword();
+fieldTesting.desireSnapshot("login");
+if (fieldTestsing.isValidationRun()) {
+  userName = fieldTesting.choice(
+       new String[] { "", "\\"", "null"},
+       "chooseUserName");
+  password = fieldTesting.choice(
+       new String[] { "", "\\"", "null"},
+       "choosePassword");
+}
```

A developer wants to check whether the program can reject illegal usernames and passwords in the production environment rather than in a clean-slate testing environment. He first uses `desireSnapshot()` to suggest generating a snapshot for subsequent field testing. Then he invokes `isValidationRun()` to check if the program is running in validation runtime and add various possible illegal values to `userName` and `password`. When running on Jaft, the program dumps a snapshot and continues its normal logic. The snapshot is then loaded into the validation runtime for field testing.

2.1.2 Decision interface. This interface lets users customize the testing space traversal strategy, with primary functions detailed in Table 1. Using these, users can implement a traversal function to control program execution in the validation runtime for field testing. The function `getCurrentChoices()` could be used to obtain all the choices available when the program execution reaches here, while `proceedWithChoice()` allows for trying a particular choice and provides some execution information through its return value.

Additionally, we need functions to retrieve the program's current state for better decision-making and to report errors. Although not listed in Table 1, these functions are part of the decision interface. Together, they simplify writing testing space traversal strategies, i.e., field-testing execution strategies.

2.1.3 Debugging interface. In our design, the field testing code related to the `choice()` function is only reachable in Jaft's validation runtime, complicating correctness checks. The debugging interface addresses this by allowing users to control field-testing code execution without the runtime module, enabling users to write unit tests for correctness checking conveniently.

While the choice interface defines a testing space by providing specific assignments for uncertainties, the debugging interface is designed to check if the execution results of one or several combinations of assignments to uncertainties are in line with users' expectations. In other words, it involves specifying a traversal path in the testing space and checking that the execution results are expected. This essentially requires controlling the program execution through managing the uncertainties, specifically return values of the `choice()` function provided by the choice interface, similar to using a debugger. Thus, we designed an interface with debugger-like functionalities, allowing users to control program execution in the development environment before deployment to Jaft's runtime module.

A program with field testing code acts as a debugging server. Users can write a debugging client using the debugging interface to control the program's execution and observe its behavior. This control and observation code can form a unit test case, providing some assurance of correctness for the field testing code.

2.2 Snapshot-based Isolation

Our field testing API design divides testing into testing space declaration and testing execution strategy, making Jaft developer-friendly. We still need a robust testing isolation mechanism to make it production-ready. On the one hand, it should offer strong enough isolation to mitigate the impact of field testing on the running program; on the other hand, it should impose small enough restrictions so that most existing testing techniques could be easily implemented on it. A program snapshot perfectly meets our requirements: snapshot can be executed in an isolated environment for testing thus minimizing the impact on the running program, and it is an exact "copy" of the running program, allowing any testing technique to be applied seamlessly.

To make the snapshot more lightweight and facilitate future optimization of snapshot generation and test execution by leveraging Java semantics, we decided to save the state of the Java program derived from the Java Virtual Machine Specification [1]. Since there has never been a definition for a Java program snapshot, defining a Java program snapshot format and determining how to restore the original program state from the snapshot have become two important problems to solve. We do not save external environment states, like opened files or sockets, for now, believing that this can already facilitate many types of tests. Modeling and saving external states could perhaps be considered as future work. Next, we will introduce our Java program snapshot design, followed by the program state migration process.

2.2.1 Java Snapshot Format. According to the definitions in the Java Virtual Machine Specification [1], the data structures related to the state of a Java program running in a Java Virtual Machine mainly include the following parts:

- *Java Virtual Machine Stack*: Java method call stack.
- *Native Method Stack*: native method call stack.
- *Method Area*: definitions of Java classes are stored in it.
- *Heap*: object data are stored in it.
- *Runtime Constant Pool*: some literals and dynamically-resolved references to methods and fields are stored in it.

These components form the runtime data area of the Java Virtual Machine, containing nearly all information about the running Java program. Below, we analyze this data to identify the information that needs to be saved in the Java program snapshot.

Java virtual machine stack and heap are dynamically updated during the program's execution and must be included in the snapshot. As for the native method stack, it is beyond the scope of Java language specifications, we do not consider it in the snapshot. Instead, we could choose an appropriate snapshot timing so that the subsequent testing based on the snapshot does not need information in the native method stack. Since the runtime constant pool could be easily reconstructed using the information in the class definition, we do not have to store it. Method area seems to be the same case — we could rebuild it using immutable class files. However, Java has a feature called dynamic class loading, allowing classes to be generated and loaded on the fly, which means class files may not always be available. So we still have to save complete class definitions obtained from the method area in the snapshot.

In summary, our snapshot primarily includes: *Java Virtual Machine Stack* for thread data, *Heap* for object data, and *Method Area* for complete definitions of classes. Additional implementation-related information may be needed to successfully restore the program state, which will be discussed in the implementation section.

2.2.2 Program State Migration. The Java program snapshot we designed could theoretically be loaded and executed in any JVM conforming to Java Virtual Machine Specification, which means our validation runtime could be based on any JVM. To achieve this flexibility and decouple the system, we have to design a general snapshot loading (state restoring) process, migrating the program state we get from one JVM to another. In this process, we need to handle some implementation differences for some Java data structures. According to our analysis, there are three main sources of differences:

- Core classes, such as `java.lang.Object`, `java.lang.Class`, `java.lang.Thread`, etc., because their implementations are often closely tied to the implementations of JVMs.
- Classes that interact with the underlying operating system, such as those related to I/O and networking, because they often invoke C/C++ native functions that are not portable.
- Certain classes a JVM implementation wants to simplify or optimize, such as HotSpot JVM's implementation of `Object.hashcode()` involves optimization using native code.

The number of such classes is limited and many of them conform to the Java SE API specification, so their differences among different Java Virtual Machines are not too significant. This provides some assurance for successfully handling them.

When loading the Java program snapshot, we must transform the Java data to handle differences. This is essentially a "dynamic software update" [26] process, where we dynamically "update" some class definitions from the Java Virtual Machine where the snapshot was created, enabling the program to continue running normally on a different Java Virtual Machine.

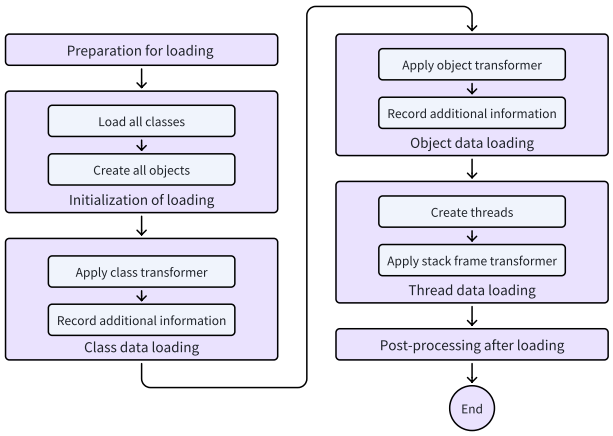


Figure 2: Snapshot loading process. It is a relatively general loading process for Java program snapshots that we propose utilizing "transformers" similar to those used in dynamic software update.

We design a general Java program snapshot loading process as shown in Figure 2 utilizing "transformers" similar to those used in dynamic software update. This loading process is primarily divided into four main stages:

(1) *Initialization of loading*: This stage primarily involves creating data structures for classes and objects to facilitate loading field values.
(2) *Class data loading*: This stage mainly deals with the static fields of classes. We use various "class transformers" to handle differences in static field definitions.
(3) *Object data loading*: This stage primarily deals with the instance fields of objects. Similarly, "object transformers" are used to deal with differences in instance field definitions.
(4) *Thread data loading*: This stage focuses on the information in stack frames. For method implementation differences, we use "stack frame transformers" to convert the data.

These transformers essentially use one data structure to construct another that is functionally equivalent but implemented differently. Usage of them helps us design an extensible loading process, which could be easily adapted by adding or removing some transformers. Of course, this loading process also leaves places for some other miscellaneous work, such as initialization before loading, post-processing after loading, and recording additional information in different stages of loading.

2.3 Automatically Adjustable Runtime Module

The previous sections introduced the programming interface design and isolation mechanism of the Jaft system. To effectively integrate these elements in the production environment, we designed a runtime module to coordinate field testing execution. The runtime module consists of three main parts: the *production runtime* for generating Java program snapshots, the *validation runtime* for running and testing these snapshots, and a *coordinator* that interacts with users and controls the entire system. The structure and workflow of the runtime module are shown in Figure 1. Next, we will introduce the runtime module's user interface and its method for automatically adjusting to meet user-defined resource limits for field testing.

2.3.1 User Interface. The runtime module's user interface is provided as a command-line tool, enabling users to start/stop field testing instances, query results, and inspect/modify configurations via the command line. This tool can seamlessly replace the `java` command, making the system easy to use. It also allows users to configure Jaft, with settings divided into "system configurations" for the entire system and "instance configurations" for each field testing instance. System configurations include CPU and memory usage limits, both expressed as percentages. Instance configurations control properties of field-testing Java programs, such as the minimum interval for snapshot generation in the production runtime, maximum execution time in the validation runtime, whether to generate trace files, and which testing space exploration strategy to use, etc. Users can leverage these configurations to manage resource consumption and enhance the effectiveness of field testing.

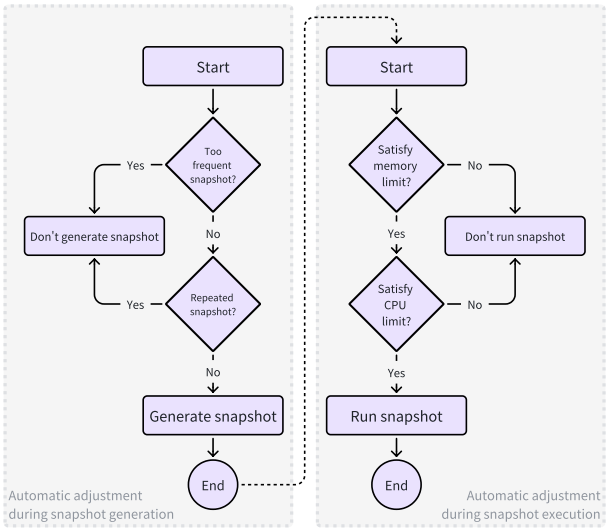


Figure 3: Automatic Adjustment Method in the Runtime Module. It is primarily achieved by regulating the frequency of snapshot generation and execution.

2.3.2 Automatic Adjustment Method. To ensure Jaft is practical, we need to minimize the impact of testing activities on the normal operation of running Java programs. The automatic adjustment method shown in Figure 3 addresses this by dynamically adjusting field testing's resource consumption. Implemented primarily by the coordinator of the runtime module, it targets user-set CPU and memory usage limits. This mechanism automatically reduces resource use when the testing load is high and increases it when the load is low, enhancing efficiency and practicality.

3 SYSTEM IMPLEMENTATION

Since our target is Java programs, we need an appropriate Java runtime, specifically a Java Virtual Machine (JVM), to implement our field testing system, Jaft. Our design requires generating and executing Java program snapshots–capabilities not available in current commercial JVMs. Therefore, we need to implement these features ourselves based on existing commercial JVMs. According to their purpose, we choose OpenJDK 11, a widely-used mainstream open-source Java runtime, as the basis of the production runtime and Java Pathfinder (JPF) [24], a JVM dedicated for validation, as a basis of the validation runtime.

Our implementation consists of two main parts, field testing API and runtime module, which consists of validation runtime, production runtime, and a coordinator:

Component	Line of Code	Language
Field Testing API	~500	Java
Validation Runtime	~5,000	Java
Production Runtime	~2,000	C++
Coordinator	~1,500	Java

We keep our intrusion to OpenJDK minimal. Only ~200 lines of code in the patch to HotSpot JVM are reachable *before* fork snapshot–all rest part of the system does not impact the function of the running production system.

3.1 Field Testing API Implementation

3.1.1 Choice Interface. It mainly includes three functions: `desireSnapshot()` function for requesting snapshots, `isValidationRun()` function to check the execution environment, and `choice()` function for providing multiple choices for uncertainties. To support both production run after deployment and debugging run in the development environment, we use a flag variable to distinguish them. We'll discuss the debugging-related logic later; here, we focus on their implementations in the production runtime. In this case, `isValidationRun()` should return `false`, and `choice()` could simply return `null`, as it won't be executed in production according to our design.

The implementation of `desireSnapshot()` function is more complex, requiring support from the underlying JVM used by production runtime. We need to modify the source code of OpenJDK 11's Java Virtual Machine, HotSpot, to generate the program snapshot while it is running a Java program. However, due to HotSpot's optimization, the complete Java program state could only be obtained when all Java threads reach "safepoint"s, which are scattered among some method returns and loop back edges. To generate the snapshot, we use HotSpot's internal mechanism designed for garbage collection and other runtime services that occur at safepoints. According to HotSpot's conventions, operations at safepoints must be encapsulated as subclasses of `VM_Operation`. Thus, we created the `VM_Snapshot` class to handle snapshot generation. In its implementation, a new process is forked to dump the snapshot, allowing the original Java program process to continue running. The snapshot generation process involves serializing the runtime data, which is accessible in the forked process. With this class, we can easily implement the `desireSnapshot()` function.

3.1.2 Decision Interface. This type of interface mainly operates in the validation runtime, based on JPF. As mentioned earlier, JPF views program

execution as state space exploration. When uncertainties arise during execution, the program may transfer to different states, and the state exploration strategy comes into play. For each type of uncertainty, JPF uses a class implementing the `ChoiceGenerator<T>` interface to record the tried and remaining assignments to uncertainties. To implement specific exploration strategies, JPF provides the abstract class `Search`, from which all exploration strategies must inherit and implement the `search()` method. The state transition functions `forward()` and `backward()` are used to continue program execution after determining a specific assignment to uncertainty and to restore the program state before the transition, respectively. However, these functions can only be executed based on the current state, which limits the flexibility of state exploration.

To isolate the complex functions in JPF and allow users to flexibly implement state traversal algorithms, we have encapsulated these mechanisms in JPF and provided a traversal method centered on the Java program's state. We created the `ValidationVMState` class to represent the state of a Java program running on JPF, with `getCurrentChoices()` and `proceedWithChoice()` as member methods. This enables users to save `ValidationVMState` objects and explore the state space from these points. To help users obtain the current state of the running Java program, we provide a series of state query interfaces, encapsulated in the abstract base class `ValidationVMSearch`. Users can implement customized state exploration strategies by inheriting this class and implementing the abstract method `void search(ValidationVMState initVMState)`.

3.1.3 Debugging Interface. The debugging interface allows users to easily write unit test cases without runtime module support, control the execution of the program under test, and check the correctness of the added field testing code. It can control the return value of the `choice()` function in the choice interface. As mentioned before, we use a flag variable to distinguish between production and debugging runs, ensuring the choice interface executes the appropriate logic. In the implementation, we start the debugging client and server in separate threads and use Java's multi-thread synchronization mechanism to pause the debugging server, i.e., the program under test, at `choice()`, and hand control to the debugging client, which will guide its execution.

3.2 Snapshot Format Implementation and Migration

Based on the Java Virtual Machine Specification, we designed a general Java program snapshot that stores all the necessary information to continue program execution. The snapshot includes the contents of the heap, stack, and method area, with its generation process essentially involving the serialization of these three data parts. We implemented the main part of our snapshot using JSON, a widely used format with many available tools for easy content querying. We store the heap data using HotSpot's heap dump format. Consequently, our Java program snapshot consists of two files: a JSON file and a heap dump file. While we aim for a JVM-independent Java program snapshot format, some issues arise from JVM differences. Here, we discuss the two most challenging issues and our solutions.

3.2.1 Handle `Unsafe` access. The `Unsafe` class allows direct read and write access to instance and static fields in Java using offsets from the start of these data structures. These offsets are platform-specific and not portable. Consequently, after a snapshot is loaded, `Unsafe` accesses using the stored offsets will cause errors. Upon analysis, we found that for each object or class, the mapping from field offset to field information is one-to-one. This means that if we know the type of the object or class and the offset, we can determine which field is being accessed. Our solution is to save the offsets of instance and static fields for each class in the snapshot. During snapshot execution, when an `Unsafe` call occurs, we use the object reference to retrieve its type information and the given offset to look up our stored mapping. This allows us to identify and directly access the specific field. However, for field offsets obtained during snapshot execution, this special handling is unnecessary and could cause issues if applied.

3.2.2 Deal with information loss of `Object.hashcode()`. In Java, the `hashcode()` method of the class `Object` is used to obtain an object's hash value, commonly used in data structures like hash tables. HotSpot, our production JVM, has an optimized implementation for the `hashcode()` method in the class `Object`. To support various hash algorithms and ensure fast access, HotSpot implements `hashcode()` at the virtual machine level and typically stores the object's hash value in the object's "header". This header is a small memory segment reserved by the virtual machine at the beginning of each object's data area, preceding the instance field values and other data. It is not a Java-level data structure and is not included in our snapshot, but we need it for the snapshot to run correctly. The challenge is that the hash code does not always reside in the object header and its location changes based on the object's synchronization state. Additionally, the hash value computation function cannot be called while the program is at a safepoint, preventing us from using this function during snapshot generation. Therefore, we retrieve the hash value from different locations based on the object's state and store each object's hash value in the snapshot.

3.3 Runtime Module

The primary function of the runtime module is to schedule the Jaft's operations and provide a command-line interface for user interaction, system configuration, and status queries. Here, we focus on the command-line interface implementation. The first time the command-line tool is used to start field testing after each OS startup, it initiates a daemon to monitor all running field-testing instances. This daemon periodically logs the CPU and memory usage of the field testing into a log file. User operations performed via the command-line tool are also logged for the daemon to analyze resource usage and for user queries. Each time a user starts field testing, default configurations are set and written into a global configuration file. Users can later query and modify these configurations through the command-line tool.

4 EVALUATION

In this section, we evaluate Jaft to answer the following questions.

(1) **RQ1**: Is Jaft's overhead tolerable?
(2) **RQ2**: Does Jaft help its users to achieve better testing results, for example, higher testing coverage?

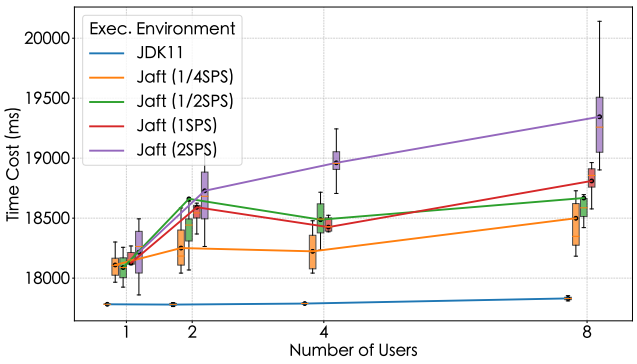
4.1 Experimental Setup

We chose Apache FtpServer[1] and H2 Database[2], two widely used and actively maintained Java software applications, as our experimental subjects to closely simulate real-world scenarios. Apache FtpServer serves as an example of an I/O-intensive application, whereas the in-memory mode H2 Database represents a memory- and CPU-intensive system.

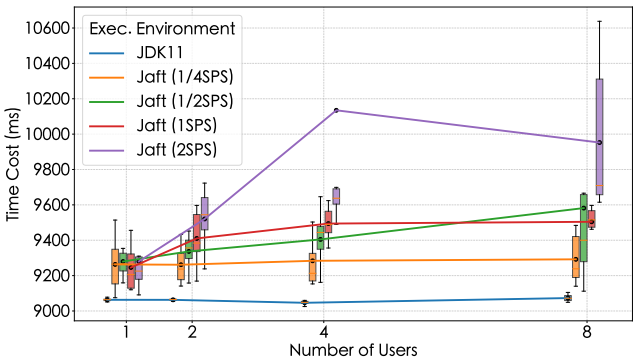
RQ1 (Overhead). We measure the overhead of Jaft on various workloads (with field testing) and compare the performance with an unmodified OpenJDK 11 build. The workloads for the FTP server are:

- `uploadDownload`: repeatedly creating multiple directories and uploading files to these directories. The user then downloads all files and removes them from the server.
- `uploadRename`: Similar to `uploadDownload`, except that it renames files and moves them out of the original directories instead of downloading them.
- `uploadResume`: Similar to `uploadDownload`, except that it resumes uploading before downloading all the files.

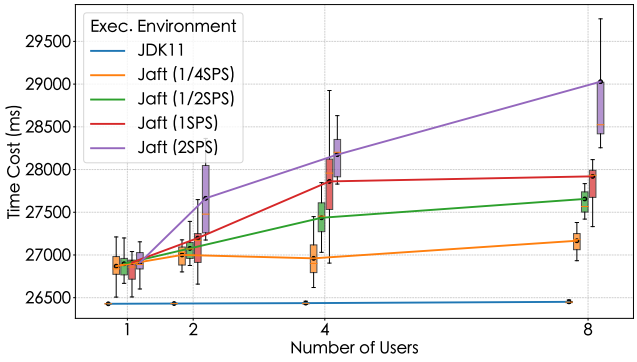
[1] https://mina.apache.org/ftpserver-project/
[2] https://www.h2database.com/html/main.html



(a) `uploadDownload`

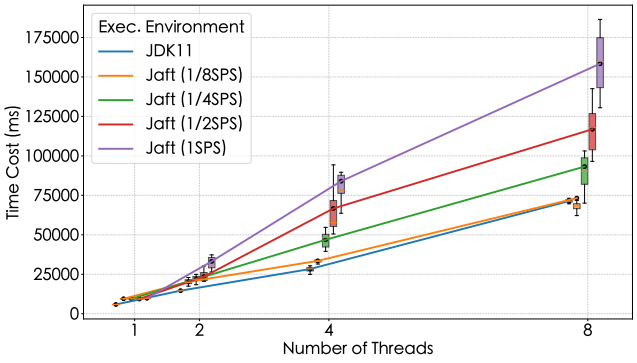


(b) `uploadRename`

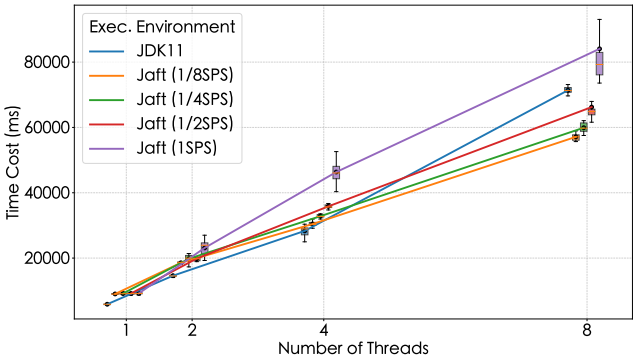


(c) `uploadResume`

Figure 4: Overhead evaluation results for Apache FtpServer.



(a) benchB with default heap limit



(b) benchB with heap limit 4G

Figure 5: Overhead evaluation results for H2 Database.

We conducted these tests with 1, 2, 4, and 8 parallel users. Apache FtpServer will assign each request a separate thread. For each experimental configuration, we calculated the average time it took for users to complete the test case as the running time. Each configuration was run independently 10 times, and the average running time was used to calculate the overhead. By adjusting the minimum time interval between snapshot generations as per Jaft's design, we were able to manage the system's overhead. We set different snapshot intervals to check if the overhead could be controlled within a tolerable range. For each configuration, we evaluate the overhead of Jaft under a maximal snapshot rate of 1/4, 1/2, 1, and 2 per second.

We ran H2 Database in its in-memory mode making it memory- and CPU-intensive to evaluate Jaft's overhead on this type of application. We chose benchB from H2 Database's test suite, which is a benchmark simulating the TPC-B and testing the database's performance under concurrent operations. We also test H2 Database with 1, 2, 4, and 8 parallel threads, and evaluate

the overhead of Jaft under maximal snapshot rate of 1/8, 1/4, 1/2, and 1 per second. The overhead calculation method is the same as described above.

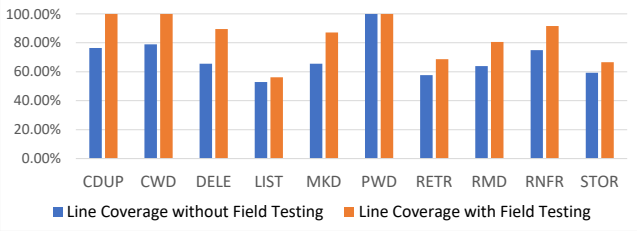


Figure 6: Comparison of code coverage for running Apache FtpServer on a practical workload with/without field testing.

RQ2 (Effectiveness). To evaluate the effectiveness of Jaft in real-world field testing. We act as if we are the developers of Apache FtpServer, and add about 150 lines of field-testing code into the repository targeting uncertainties brought by user inputs and file systems. We used field testing API to explore users' less common program paths, e.g., invalid or empty commands. We also specify points of fault injection by throwing an exception on file system operations to simulate failure. We applied Jaft to conduct field testing regarding these cases. Most of the added code is only reachable when field testing is enabled, and thus does not affect the FTP server's behavior. To simulate a real-world workload, we have selected 10 commonly used FTP server commands based on our experience:

- Directory operations: create a directory, get the current directory, enter a directory, go to the parent directory, delete a directory, and list files in a directory.
- File operations: upload a file, download a file, rename a file, and delete a file.

We design a test case simulating a user that performs these actions with a 1-second delay and collect the code coverage of Apache FtpServer for both with and without field testing to assess the effectiveness of Jaft.

Environment. All experiments were conducted on a workstation of a single 32-core (64-thread) AMD Ryzen Threadripper PRO 5975WX processor and 128 GB RAM running Ubuntu 22.04 LTS.

4.2 RQ1: Overhead

The overhead experimental result of FTP benchmark test case `uploadDownload`, `uploadRename`, and `uploadResume` is detailed in Figure 4. For each configuration of benchmark, we run it 10 times and draw the box graph to display the running time across various execution environments and user counts. As for the execution environment, "JDK11" means the benchmark runs on unmodified OpenJDK 11, and others represent execution on Jaft with different maximum snapshot generation rates, where "SPS" means "snapshot per second". For example, "1/2SPS" means at most one snapshot is generated every two seconds.

Apache FtpServer scales well: the running time of Apache FtpServer on JDK 11 is not significantly impacted over increased parallelism. This is expected because each request is handled in a separate thread and there is minimal data sharing between threads. However, when running on Jaft, an increase in the number of users leads to a gradual increase in execution time. This is because the snapshot mechanism of Jaft can only be conducted on VM safepoints, which requires all threads to be synchronized. The increase in the number of users leads to more threads, thereby increasing the synchronization overhead and execution time. We could also clearly observe that as the snapshot generation rate decreases, the overhead introduced by Jaft gradually decreases. Following the criteria from the referenced study [8], an overhead of 5% in the production environment is tolerable. Our results show that we could achieve this by adjusting the snapshot generation rates of Jaft so it could be deployed in the production environment.

The results for H2 Database benchmark `benchB` are presented in Figure 5a. For H2 Database, an increase in the number of threads significantly affects the runtime for all cases, as these threads access the same set of data within the database, leading to increased synchronization overhead and longer execution times. It is also observed that under the same snapshot generation rate, the overhead imposed by Jaft on H2 Database is considerably greater than that on Apache FtpServer. However, it is noted that as the snapshot generation rate decreases, the overhead from Jaft also gradually decreases, indicating the effectiveness of our control strategy.

The reason H2 Database incurs greater overhead on Jaft compared to Apache FtpServer can be attributed to two factors. Firstly, as a CPU-intensive case, the synchronization required for snapshot generation delays execution, whereas Apache FtpServer spends more time waiting and handling IO operations, which mitigates the impact of synchronization delays caused by snapshot operations. Secondly, as an in-memory database, H2 Database likely has a larger heap size, leading to more frequent occurrences of page faults triggering after the "copy-on-write" fork, further delaying execution. Additionally, our server's large memory capacity allows the Java Virtual Machine to set a high heap limit, enabling H2 Database to run with a significantly large heap size during execution, which exacerbates the seriousness of the problem.

Therefore, we conducted a second experiment using the Java Virtual Machine option "-Xmx" to manually set the heap limit of both JDK 11 and Jaft to 4G, while keeping other settings unchanged. It was demonstrated that the benchmark could also successfully run using less than 4G of heap, and the results were shown in Figure 5b. We noted that with a smaller heap size, the overhead introduced by Jaft was indeed reduced, which confirmed our above analysis. This implies that Jaft exhibits a significant performance impact with larger heap sizes, and we need to adjust the snapshot generation rate to smaller values to mitigate the incurred overhead.

Conclusion. Experiments conducted on the practical Java applications Apache FtpServer and H2 Database demonstrate that by adjusting the snapshot generation rate in Jaft, we could manage the system overhead. Moreover, we can maintain this overhead within a tolerable limit for production environments (within 5%), making it feasible for deployment in production settings.

4.3 RQ2: Effectiveness

We designed test cases to simulate real user interactions with an FTP server and analyzed the line coverage of the main functional classes involved. The results, displayed in Figure 6, compare the line coverage between Apache FtpServer running on JDK 11 and Apache FtpServer tested in a production environment on Jaft. Since all these classes are located in the same package, the package name is omitted in the figure.

According to Figure 6, there is an overall improvement in coverage for these classes, with an average increase in line coverage of 14.48%. The coverage for the class `PWD` remained unchanged because it already achieved 100% code coverage on JDK 11 without the field testing.

Conclusion. The experiments simulating real user scenarios on Apache FtpServer demonstrate that developers or testers can utilize the Jaft to write test codes for production environments. By leveraging their knowledge of the code and the richer inputs and various scenarios available in production, they can enhance test coverage thus achieving better testing results.

5 RELATED WORK

Researchers have recognized the severity of the problem that the bug escapes in-house testing causing field failures [11, 12, 23] and tried to figure out why. Gazzola et al.[13] conducted an in-depth study of field failures over three

enterprise-level applications. They found that 70% of them are intrinsically hard to find by in-house testing and proposed field testing as a promising direction to mitigate this problem. Bertolino et al.[4] also identified field testing as an effective way to improve current quality assurance practice and presented a systematic survey of existing field testing techniques. Here we introduce some related research work of field testing.

Continuous quality assurance. Residual testing [20, 22] monitored software execution in the production environment to collect some data helping developers to improve software quality. Gamma [21] went one step further. It divided the monitoring tasks and distributed them to different running instances. Skoll [18] distributed testing instead of monitoring tasks to deployed software. QVM [2] modified the Java Virtual Machine to implement lightweight monitoring in the production environment.

Field testing. Component-based systems and distributed systems are more prone to face insufficient testing problems due to their complexity. Researchers have tried to develop field testing techniques targeting them. Lahami et al. proposed RTF4ADS [15–17] using field testing to solve the testing problem of component-based software due to its structural adaptation during running. CrytalBall [25] used distributed snapshots to conduct field testing on running distributed systems and tried to stir them away from error states. In-vivo testing [5, 7, 10, 19] is a general field testing technique targeting Java software. This type of work requires users to prepare test cases called in-vivo tests, which are similar to unit tests but have to pass regardless of the program's running state. Different work of this type may use different isolation strategies. For example, Invite [19] used fork, StealthTest [7] used transaction memory, while Groucho [5] took advantage of an existing checkpointing-and-rollback technique implemented for Java [3]. Though being a general technique, in-vivo testing needs users to provide a special type of test case, which is not easy to write. Their isolation mechanism is either not robust enough to use in production or not powerful enough to support various testing methods.

6 CONCLUSION

In this paper, we design and implement a field testing system Jaft. It provides users with field testing API for them to easily customize testing strategies. It uses snapshot-based isolation to decouple the production system from testing activities to make the system robust. And the runtime module of it allows users to adjust the configurations of the system dynamically thus making the system efficient.

Evaluation of Jaft shows that the runtime overhead of the system could be set to low enough to be used in a production environment and it could help users to improve the coverage of the SUT thus getting better testing results. Overall, it is a practical field testing system that has the potential to be widely adopted to improve the current software quality assurance situation.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China under Grants No. 61932021, No. 62025202, No. 62272218, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn), the corresponding author, was support by the Xiaomi Foundation.

REFERENCES

[1] 2021. Java Virtual Machine Specification, Java SE 21. https://docs.oracle.com/javase/specs/jvms/se21/html/index.html Accessed: 2024-03-07.

[2] Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, Nashville TN USA, 143–162. https://doi.org/10.1145/1449764.1449776

[3] Jonathan Bell and Luís Pina. [n. d.]. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. ([n. d.]), 31 pages, 643549 bytes. https://doi.org/10.4230/LIPICS.ECOOP.2018.17 Artwork Size: 31 pages, 643549 bytes ISBN: 9783959770798 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. 2022. A Survey of Field-based Testing Techniques. *Comput. Surveys* 54, 5 (June 2022), 1–39. https://doi.org/10.1145/3447240

[5] Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Paolo Tonella. [n. d.]. In vivo test and rollback of Java applications as they are. 33, 7 ([n. d.]), e1857. https://doi.org/10.1002/stvr.1857

[6] Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Paolo Tonella. 2020. Run Java Applications and Test Them In-Vivo Meantime. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 454–459. https://doi.org/10.1109/ICST46399.2020.00061

[7] Jayaram Bobba, Weiwei Xiong, Luke Yen, Mark D. Hill, and David A. Wood. 2009. StealthTest: Low Overhead Online Software Testing Using Transactional Memory. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Raleigh, North Carolina, USA, 146–155. https://doi.org/10.1109/PACT.2009.15

[8] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A deployable sampling strategy for data race detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 810–821. https://doi.org/10.1145/2950290.2950310

[9] Mariano Ceccato, Davide Corradini, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orru, and Paolo Tonella. 2020. A Framework for In-Vivo Testing of Mobile Applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 286–296. https://doi.org/10.1109/ICST46399.2020.00037

[10] Matt Chu, Christian Murphy, and Gail Kaiser. 2008. Distributed In Vivo Testing of Software Applications. In *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, Lillehammer, Norway, 509–512. https://doi.org/10.1109/ICST.2008.13

[11] Wikipedia contributors. 2024. *2011 Southwest blackout*. https://en.wikipedia.org/wiki/2011_Southwest_blackout Accessed: 2024-03-06.

[12] Wikipedia contributors. 2024. *Heartbleed*. https://en.wikipedia.org/wiki/Heartbleed Accessed: 2024-03-06.

[13] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. 2017. An Exploratory Study of Field Failures. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Toulouse, 67–77. https://doi.org/10.1109/ISSRE.2017.10

[14] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, San Diego California USA, 179–188. https://doi.org/10.1145/1250734.1250755

[15] Mariam Lahami. 2017. *Runtime testing of dynamically adaptable and distributed component based Systems*. PhD Thesis. École nationale d'ingénieurs de Sfax, Tunisia. https://tel.archives-ouvertes.fr/tel-02469999

[16] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. 2013. Runtime testing framework for improving quality in dynamic service-based systems. In *Proceedings of the 2013 International Workshop on Quality Assurance for Service-based Applications*. ACM, Lugano Switzerland, 17–24. https://doi.org/10.1145/2489300.2489335

[17] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. 2016. Safe and efficient runtime testing framework applied in dynamic and distributed systems. *Science of Computer Programming* 122 (June 2016), 1–28. https://doi.org/10.1016/j.scico.2016.02.002

[18] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. 2004. Skoll: distributed continuous quality assurance. In *Proceedings. 26th International Conference on Software Engineering*. IEEE Comput. Soc, Edinburgh, UK, 459–468. https://doi.org/10.1109/ICSE.2004.1317468

[19] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. 2009. Quality Assurance of Software Applications Using the In Vivo Testing Approach. In *2009 International Conference on Software Testing Verification and Validation*. IEEE, Denver, CO, USA, 111–120. https://doi.org/10.1109/ICST.2009.18

[20] Leila Naslavsky, Marcio S. Dias, and Debra J. Richardson. 2004. Multiply-deployed residual testing at the object level. In *IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 17-19, 2004*, M. H. Hamza (Ed.). IASTED/ACTA Press, 396–401.

[21] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. 2002. Gamma system: continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, Roma Italy, 65–69. https://doi.org/10.1145/566172.566182

[22] Christina Pavlopoulou and Michal Young. 1999. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 277–284. https://doi.org/10.1145/302405.302637

[23] Pak-Lok Poon, Man Fai Lau, Yuen-Tak Yu, and Sau-Fun Tang. 2024. Spreadsheet quality assurance: a literature review. *Frontiers Comput. Sci.* 18, 2 (2024), 182203. https://doi.org/10.1007/S11704-023-2384-6

[24] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model checking programs. *Automated software engineering* 10 (2003), 203–232.

[25] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 229–244. http://www.usenix.org/events/nsdi09/tech/full_papers/yabandeh/yabandeh.pdf

[26] Zelin Zhao, Yanyan Jiang, Chang Xu, Tianxiao Gu, and Xiaoxing Ma. 2021. Synthesizing Object State Transformers for Dynamic Software Updates. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1111–1122. https://doi.org/10.1109/ICSE43902.2021.00103