Alpen-Adria-Universität Klagenfurt
Software Engineering Research Group
Technical Report Series

Automatic Identification and Recovery of Errors in Programs - A Rigorous Approach

Giovanni Liva, Francesco Spegni, Muhammad Taimoor Khan, Luca Spalazzi, and Martin Pinzger

Report AAU-SERG-2019-002

AAU-SERG-2019-002

Published, produced and distributed by:

Software Engineering Research Group
Institute of Informatics Systems
Faculty of Technical Sciences
Alpen-Adria-Universität Klagenfurt
Universitätsstraße 65-67
9020 Klagenfurt
Austria

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://serg.aau.at/bin/view/Main/Publications

For more information about the Software Engineering Research Group:
http://serg.aau.at

Automatic Identification and Recovery of Errors in Programs - A Rigorous Approach

Giovanni Liva*, Francesco Spegni†, Muhammad Taimoor Khan‡, Luca Spalazzi†, and Martin Pinzger*
*Software Engineering Research Group, Alpen-Adria Universität Klagenfurt, Austria
Email: {giovanni.liva, martin.pinzger}@aau.at
†Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche, Italy
Email: {f.spegni, l.spalazzi}@univpm.it
‡Department of Computing and Information Systems, University of Greenwich, UK
Email: m.khan@greenwich.ac.uk

***Abstract*—Modern software development practices have shifted to a fast and agile release process. Software is automatically compiled, tested, verified, and shipped to the users. Formal methods and their implementation in tools are difficult to integrate into such a process. Existing techniques try to close this gap by providing static analysis tools that can be run along with the test harness. However, these approaches require a formalization of the full system. This is not always possible since the environment, in which the program is run, is not known a priori. In this paper, we present a technique that combines static and dynamic analysis to define formal tools that are amenable to automatically recover the program state at runtime when errors are detected. Our proposed approach introduces an Assertion Library and an Assertion Plugin. The library enables developers to assert properties of their program and to verify them at runtime. The plugin is used by the build system of a program to generate an abstract model of the source code and attach it to the artifact produced. Furthermore, during the generation of the model, it verifies that no violations of the assertions are identified. When a violation is found either at compile- or run-time, a meaningful counter-example is presented. This can be used by developers to improve the code in the next release cycle. Furthermore, our approach tries to automatically recover the correct system behavior based on the violated assertion. For the evaluation, we have created a dataset that consists of 100 bugs collected from 20 open source Java projects. We use the dataset to verify the ability of our approach to identify and recover from the errors. The results of our experiments show that our approach is able to identify 100% of the errors and automatically recover 84% of them. Furthermore, the runtime overhead introduced by our approach has a median value of 119.5ms.**

I. Introduction

Due to the fast pace of the software market, companies need to release new features and updates frequently and fast to stay competitive. Furthermore, developers need to analyze and monitor the usage of their software products to derive data that can be used to improve the software and increase their revenue stream. Therefore, the process used to develop software has switched to fast and agile techniques. These techniques aid developers to automate some of their activities with the benefit of allowing them to focus on implementing new features. For this reason, software companies endorse the automation of building, testing and deploying software systems, such as done with modern continuous delivery and continuous integration (CI/CD) pipelines.

Formal methods and their implementation in tools are difficult to integrate into such a fast and mostly automated development process. With the current formal techniques, such as [1]–[6], developers have to manually define invariants and properties of their code and model them in other languages and/or tools that cannot easily be integrated into CI/CD pipelines. This affects the productivity of the software development teams and it may discourage them to apply formal verification techniques. Instead, most teams rely only on testing to verify the correct behavior of a system. Existing researches, such as [7]–[10], try to close this gap by using static analysis [11] to automatically build and verify an abstract model of the code. This type of techniques perform a symbolic execution or an abstract interpretation of the source code to automatically find specific types of errors or code smells in the implementation [12]. Big software companies, such as Google [13], [14], Facebook [15], [16], and Microsoft [17], [18] developed multiple static analysis tools to use in their software release pipelines. These techniques extracts models from the code that they use to generate tests that cover different types of properties. For instance, PEX [3], developed by Microsoft, collects path constraints with a static analysis and uses them to generate tests to achieve a high branch coverage. Similarly, concolic testing [19] performs symbolic execution of the code to generate tests that identify data race errors. However, existing static analysis approaches have some drawbacks. First, they fail to fully model the runtime behavior of the code and second, they require a formalization of the full system, which is not always feasible. With the current agile development and microservices architecture, it is not always feasible to a priori formalize the environment in which a program is run or the external libraries used, making such approaches ineffective [20]. Therefore, other formal techniques, *e.g.*, [21]–[23], focus on protecting the application at runtime. Developers define the conditions in which their software works and these techniques monitor the application at runtime to discover (and block) any deviation from the specified behavior also in case of unforeseen situations due to the environment. However, they fail to recover the correct program's execution behavior.

For instance, Listing 1 presents an excerpt of the method `poll` taken from the Apache Kafka source code that is

```
1 public void poll(long timeout) {
2   long now = time.milliseconds();
3   long deadline = now + timeout;
4   while (now <= deadline) {
5     ...//important code which is expected to
6        //be executed at least once
7   }
8 }
```

Listing 1: Source code of the method `poll()` from the class `WorkerCoordinator` of the Apache Kafka project.

responsible for the issue KAFKA-4290.[1] If the method is called with (i) any negative number or (ii) a big enough positive number, its execution will cause a failure in the program because the content of the while loop is expected to be executed at least once. With the existing techniques, if one of the aforementioned conditions occurs, the application stops its execution with an error without pinpointing the root cause that triggered the error, *i.e.*, the wrong value for the `timeout` variable. More importantly, they do not provide any self-healing mechanisms that automatically recover the program from the error, requiring developers to fix the problem in the next release cycle.

In this paper, we present a technique that overcomes the aforementioned limitations combining static and dynamic analysis to aid developers assuring runtime properties of their programs. Our technique automatically injects a verification framework into the executable of the system that discovers errors at both, compile- and run-time. The major advantage in contrast to existing approaches is that it enables the program to automatically recover from the errors that could happen at runtime. Our approach is based on two different artifacts, namely an Assertion Library and an Assertion Plugin. The Assertion Library enables developers to assert properties directly in the source code that must hold during the program execution. Furthermore, developers can combine symbolic and concrete values of variables while defining their invariants. First, the Assertion Plugin creates an abstract model of the source code and attaches it to the executable produced by the build system. Second, it verifies all the properties defined by the developers and stops the build process if it identifies that some properties cannot be satisfied in the given source code. Finally, if no violations are identified, it inserts a verification engine into the executable that verifies the properties at runtime. The runtime verification can be configured with filters that enable or disable the verification of specific properties. Compared to alternative techniques, such as Java PathFinder [4] or Larva [24], developers do not require a special Java Virtual Machine (JVM) to run the verification. They can ship their artifacts with the verification of such properties directly to the end users. Moreover, in contrast to existing similar techniques, such as JML [1], our approach is able to automatically provide a recovery strategy to resume the expected execution behavior when an error has been detected at runtime. This favors the end-user to not suffer from errors that can be fixed by the developers in the next release cycle. Furthermore, we provide the possibility to define invariants using both, symbolic and concrete variables, whereas other approaches, such as JML [1] or ESC/Java2 [25], uses only symbolic variables at compile time and concrete variables at runtime.

We have implemented our approach in a prototype tool and we have applied it to a dataset of 100 bugs collected from 20 open source Java projects. In our evaluation, we answer the following three research questions:

RQ1: Is our approach adequate to identify errors in the source code?
RQ2: How many errors can the recovery strategy solve?
RQ3: How much runtime overhead does our approach introduce?

The results of our evaluation show that our prototype manages to discover all errors from the given specifications and recovers 84% of them, requiring in median 119.5ms to identify and repair an error.

In summary, this paper makes the following contributions:

- A recovery strategy to assure a correct runtime behavior.
- An approach to design formal tools for modern software development techniques.
- A dataset of 100 bugs collected from 20 open source Java projects.

The remaining of the paper is organized as follows: Section II details our approach and we present the implementation details in Section III. In Section IV, we evaluate the approach and we discuss implications, as well as threats to the validity of our experiments in Section V. Section VI gives an overview of the related works and Section VII concludes the paper.

II. Approach

In this section, we present our approach to identify and recover errors in programs. Figure 2 shows an overview of our approach consisting of three steps. In the first step, the developers write the code of their application along with its specification using our Assertion Library. Upon committing their work into the repository, the CI/CD pipeline starts. In the second step, the Assertion Plugin intecepts the compilation of the source code and creates an abstract model of the code that is integrated into the final artifact. The Assertion Plugin performs a symbolic execution using the abstract model of the code to verify whether the specifications expressed by the developers are valid in the source code. Finally, a final artifact is produced and it contains the program, the abstract model of the source code, and the code that monitors the specifications expressed by the developers at runtime. When a property does not hold, the developers are notified with the cause that invalidated the specifications. Furthermore, developers can enable the recovery strategy offered by our approach to resume a correct program execution when errors are detected at runtime.

[1] https://issues.apache.org/jira/browse/KAFKA-4290

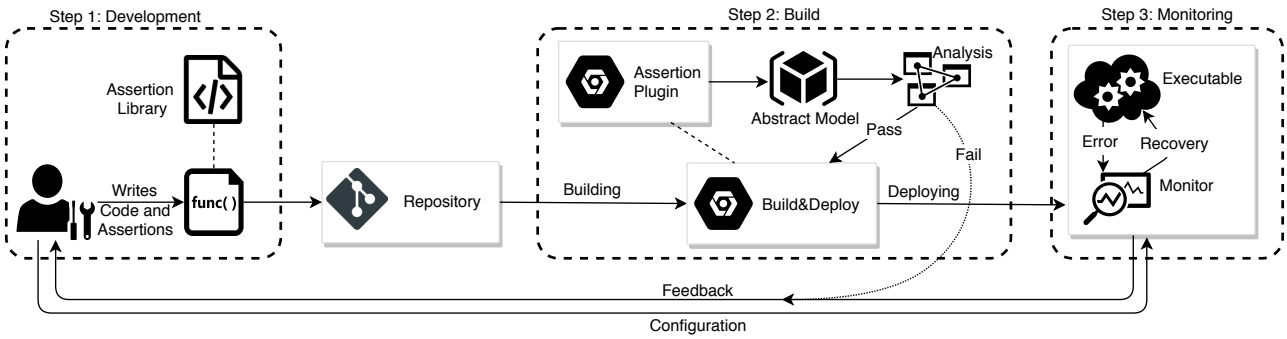

Fig. 2: Overview of our approach

```
1  public void poll(long timeout) {
2    long now = time.milliseconds();
3    long deadline = now + timeout;
4    //for all value of timeout, now must be <=
         deadline at this execution point
5    AssertLibrary.ensure(
6      forAll("timeout", leq("now", "deadline"))
7    );
8    while (now <= deadline) {
9      ... //important code which is expected to
10         //be executed at least once
11   }
12 }
```

Listing 2: Extension of Listing 1 with an assertion of the expected behavior.

A. Development Step

The first step requires to slightly change the normal workflow of a developer. Usually, developers write the code and the test harness of a given feature. Existing testing frameworks, such as JUnit or NUnit, offer an assertion interface where developers can assert conditions (*i.e.*, invariants) on the result of a method call or on the value of a variable. If the condition does not hold, an exception is thrown stopping the execution of the tests with an error message. Our approach requires developers to complement (or replace) the test harness with assertions that state the properties of a feature directly in the source code. This makes the assumptions taken by the developers when they write the code explicit. As a side effect, this can also help the documentation of the source code [26].

Multiple logics can be employed to express different types of code properties, such as First Order Logic (FOL) [27], Linear Temporal Logic (LTL) [28], or Past/Future Time LTL (ptLTL/ftLTL) [29]. Our approach provides an Assertion Library that gives developers an interface to express their specifications with different logics. The library allows to declare both, symbolic and concrete variables in the formulae allowing a more precise analysis [30], [31]. A symbolic variable is a variable that represents a set of possible values for that variable, rather than a single one. A concrete variable, instead, represents the precise value that is stored in the variable at some point in the execution of the program. Our approach considers variable names between quotation marks as symbolic variables and without quotation marks as concrete variables.

For instance, Listing 2 extends the example of Listing 1 with a formula in FOL which asserts the implicit property that the while loop should be executed at least once. For declaring this fact, a developer could write a formula which asserts that the while-condition is always verified before entering the loop. In our example, between Line 5 and 7, we express that for each value of the `timeout` parameter, the value of `now` must be less or equal to the value of `deadline`. For this assertion, only symbolic variables are used.

B. Build Step

After the changes are stored in the repository, the CI/CD pipeline starts to build and deploy the application. In this step, the Assertion Plugin intercepts the compilation of the source code and generates an abstract model of it. The model is then integrated into the final executable that is deployed in production. Furthermore, it injects a formal verification framework that monitors the application to verify the assertions of the developers at runtime. The Assertion Plugin performs an abstract interpretation and a symbolic execution of the code to generate models amenable to the verification of the different formulae expressed by the developers using the Assertion Library. Depending on the family of the formulae used in the assertions, different models of the code can be created to verify their satisfiability, such as Kripke structure, SMT problems, or automata. Furthermore, when the testing phase is executed by the build system, the Assertion Plugin verifies the formulae written by the developers. For the verification, it negates the assertion and checks its validity in the model. If the negation is satisfiable in the model, it means that the assertion is not valid in the source code and therefore, the build is halted. Along with the error information of which assertion failed, the Assertion Plugin also reports the counter-example that falsified the assertion. The counter-example helps developers to spot the cases in which the program fails.

When the Assertion Plugin processes the code presented in Listing 2, it recognizes that the formula at Line 6 specifies some properties and, therefore, it verifies it. First, it negates

the formula rewriting it from $\forall timeout.(now <= deadline)$ to $\exists timeout.(now > deadline)$. Then, it verifies the satisfiability of the negation in the abstract model generated from the source code. Note that the formula ranges over the $timeout$ variable, which is a symbolic variable that is not used in the remainder of the formula. Using the abstract model of the code during the verification of the formula, each variable is internally unrolled with its own definition collected through the symbolic execution. This makes explicit that changing the value of `timeout` changes the value of `deadline`, *i.e.*, the following formula is verified:

$$\forall timeout.\forall now.\forall deadline.(\underbrace{now \geq 0}_{\text{from time.milliseconds()}} \wedge \wedge\ deadline = now + timeout \wedge now \leq deadline)$$

Since the negation of the formula holds in the model, the build is halted and an error message is reported to the developers. Furthermore, a counter-example is reported that helps the developers understand the situations in which the code fails their assumptions. For instance, in the example of Listing 2 the following assignment is reported:

`timeout`	=	-1
`now`	=	1
`deadline`	=	0

C. Monitoring Step

If the application passes the build, it is deployed and executed. Since different environments might have different requirements, which properties are verified at runtime must be configurable on demand by the developers. Therefore, the runtime verification engine inserted into the executable can be configured with filters, via parameters or environmental variables, when the application is launched. When the execution of the application reaches an assertion, the monitor first verifies whether the assertion is eligible to be processed in accordance with the configured filters. If so, it feeds the concrete values of the variables into the formula. Then, the verification is performed with the negation of the asserted formula, as explained in Section II-B. If the assertion does not hold, the counter-example is generated and an exception is thrown by the verification engine.

If the developers enable the recovery strategy offered by our approach, the code is internally (and automatically) rewritten and the exception is not thrown. To resume a correct program execution, the recovery strategy tries to change the memory state, *i.e.*, it changes the values stored into the variables, with values that satisfies the expected execution behavior. To generate the memory values for all the variables that appear in the formula, the recovery strategy weakens the specification used by the developers. Since the original specification cannot be respected by the current state of the program, the recovery strategy relaxes the formula iteratively in several steps. At each step, the recovery strategy relaxes the formula replacing one universal quantifier with an existential one and then it verifies if the new formula can be satisfied. This approach iterates until either, the relaxed formula is satisfiable or the formula cannot be relaxed any further. Although this relaxation changes the specification, the results are an over-approximation of the original specification, *i.e.*, we admit more behavior than those possible, which should still yield a state in which the program can successfully progress its execution. In the case that also the relaxed formulae cannot be satisfied, an exception is thrown preventing the program to further continue with an invalid state.

```
1  public void poll(long timeout) {
2    timeout = timeout < 0 ? 0 : timeout;
3    long now = time.milliseconds();
4    long deadline = now + timeout;
5    try {
6      AssertLibrary.ensure(
7        forAll("timeout", leq(now, "deadline"))
8      );
9    } catch(AssertionLibraryException ex){
10     deadline = ex.aCorrectValueFor("deadline");
11     ...
12   }
13   while (now <= deadline) {
14     ...//important code which is expected to
15        //be executed at least once
16   }
17 }
```

Listing 3: Extension of Listing 2 with an example of how the code is internally rewritten by the recovery strategy.

For example, Listing 3 extends Listing 2 preventing the `timeout` parameter to be assigned with a negative value, in Line 2. This only covers the first case in which the method exposes the issue KAFKA-4290. In this example, when the monitor verifies the assertion, it returns a counter-example that shows the second erroneous case, *i.e.*, the possibility of an overflow, in which the method exposes the known issue, such as the following:

`timeout`	=	9223372036854775807
`now`	=	1542616942559
`deadline`	=	-9223370494237833250

If the developers have enabled the recovery strategy, the code is automatically rewritten similarly to the one presented between Lines 5 and 12. The verification of the formula is encapsulated in a try-catch block. Here, the formula uses the variable `now` as a concrete variable. Therefore, it computes the satisfiability of formula using the value held by the variable `now` at that execution point instead of using a symbol as its value. The catch block, in Line 9, tries to resume a correct state of the program invoking the relaxing procedure aforementioned that searches for a valid value for the variables. In this example, the recovery strategy relaxes the formula from $\forall timeout.(now <= deadline)$ to $\exists timeout.(now <= deadline)$, which is satisfiable with the following assignments:

`timeout`	=	1
`now`	=	1542616942559
`deadline`	=	1542616942560

In Line 10, an example of the recovery for the `deadline` variable is presented. Here, the variable is assigned with the value found using the recovery strategy, namely 1542616942560. In the case where no values can be found, an exception is thrown stopping the program execution.

III. Implementation

In this Section, we present the details of the implementation of our approach, which is publicly available [**?**]. Since we use Java as target language, we decided to define the Assertion Plugin in terms of an Apache Maven[2] plugin, a well known build system for Java projects. Furthermore, we decided to provide an approach to automatically repair programs based on state repair. In contrast to behavioral repair, our approach do not alter the source code of the program to remove the error. State repair automatically modifies the execution state at runtime to remove the error. Using the categorization presented by Monperrus [32], our implementation uses abstract behavioral models as oracle to identify bugs at runtime and uses a rollfoward (or forward recovery) strategy to repair the state.

Our approach relies on the implementation of the approach presented in [8] to extract the abstract behavioral models that are used to identify the errors. Although existing similar approaches, *i.e.*, JML [1] and ESC/Java2 [25], creates a behavioral model of the code at runtime, they only reason over concrete values limiting their expressiveness power. For this reason, we decided to do not extend them but and develop our own toolset. This also represent an interesting future direction for research where the Pre- and Post conditions, offered by these approaches, are combined with abstract behavioral models extracted from an abstract interpretation of the code. For the implementation of the forward recovery strategy, we relax the assertion that violates a property in the abstract behavioral models to find a new state that could resume a correct program execution. Although we performed an over-approximation of the original intended behavior, other researches, such as Sidiroglou *et al.* [33], have shown that this approach can avoiding a complete crash of software applications.

In Figure 5, we present an architecture overview of our implementation. The source code is annotated by developers with calls to the Assertion Library that express the assertions of the expected behavior, as presented in Listing 2. When the build system starts the code compilation, the Assertion Plugin starts the analysis of the source code to reverse engineer the abstract behavioral models. For this task, we used the approach presented in [8], which automatically reverse engineers SMT models from Java code. We slightly modify the approach to also model integer operations. In this extension, we provide a further mapping between mathematical operations over integer variables and SMT theories, which follows straightforwardly the approach presented in Algorithm 1 of [8]. After the analysis, the extracted models are marked as resources to be included in the final executable. Then, the build system proceeds to compile the code producing the *.class* files. In this step, the compilation requires the Assertion Library to resolve the assertion calls introduced by the developers. Since the library exports method calls that accept specific parameter types that can be called only in a specific context, we exploit the Java compiler to verify that the formulae are well-formed, *i.e.*, syntactically correct. After the compilation, the build system starts the packaging phase where *.class* files and resources are packed inside a jar file. In this phase, the Assertion Plugin rewrites the bytecode of the classes introducing our repair strategy. It looks for the method calls to the functions exported by the Assertion Library and rewrites them linking the call to the correct abstract behavioral model to use at runtime to verify the given assertion. The method `ensure()` of the Assertion Library is responsible to translate the formula in input into a First Order Logic (FOL) formula. Moreover, this method uses Z3 version 4.8.3 to verify the FOL formula holds onto the model. Finally, the Assertion Plugin rewrites the calls to the Assert Library functions encapsulating them with a try/catch block, as depicted in Listing 3. Afterwards, the build system proceeds and finishes the packaging of the application producing the executable which is comprehensive of our repair approach.

[2] https://maven.apache.org/

IV. Evaluation

In this section, we present the experiments we have performed to evaluate our approach. They aim to answer the following three research questions:

- **RQ1**: Is our approach adequate to identify errors in the source code?
- **RQ2**: How many errors can the recovery strategy solve?
- **RQ3**: How much runtime overhead does our approach introduce?

The next subsections describe the setup of the experiments and present the results for each research question.

A. Setup

We used the implementation of our approach to evaluate a dataset that comprises 100 errors mined from 20 open source Java projects. We used the same project selected for the studies presented in the previous chapters and, therefore, the projects differ in vendor, size, and domain of use. Table I presents some descriptive statistics of the selected projects. Over all projects, we selected the 100 errors from 1,064,279 distinct Java methods out of which $466,218$ contain a time related operation and $518,392$ contain an integer related operation. We selected 50 time related errors and 50 integer related errors. The 50 time related errors have been randomly selected from the dataset of error presented in Chapter **??**. The selection of the remaining 50 integer related errors was performed starting from selecting all the classes that contain at least one operation over integer variables. From this set of integer operations, we randomly selected the classes that contain those and manually verified the code. We iterated this process until we identified 50 methods that do not correctly handle integer operations.

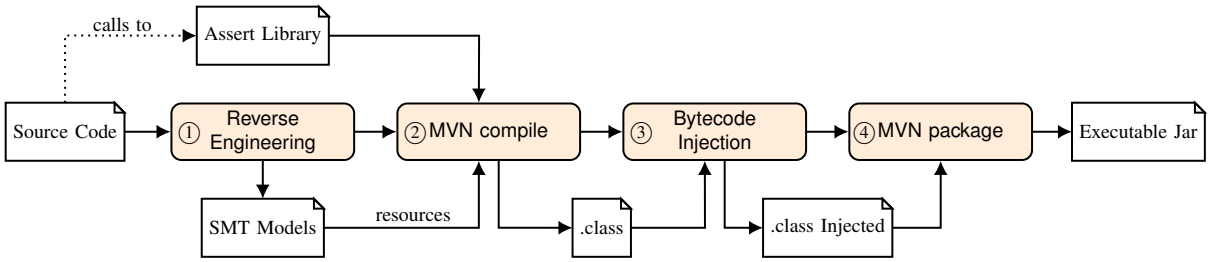

Fig. 5: Architectural overview of the implementation of our approach.

TABLE I: List of Java projects used to generate our dataset together with their number of methods, time methods, integer methods, and number of bugs.

Name	# Methods	#T. Methods	#I. Methods	# Bugs
ActiveMQ	54,026	12,583	27,048	18
Activiti	15,373	6,034	5,631	0
Airavata	73,864	39,858	24,361	3
Alluxio	25,197	13,570	12,802	4
Atmosphere	4,106	1,626	1,298	0
AWS-SDK-Java	205,438	150,932	74,175	0
Beam	31,989	7,832	15,193	0
Camel	129,811	34,760	43,329	8
Elastic-Job	2,493	637	353	0
Flume	6,862	2,429	3,854	4
Hadoop	171,189	40,173	119,166	23
Hazelcast	59,595	20,741	26,249	10
Hbase	129,405	81,747	99,540	13
Jetty	25,179	8,057	11,535	8
Kafka	14,129	5,158	7,418	0
Lens	10,614	3,917	3,426	0
NanoHTTPD	710	205	331	0
Neo4j	61,631	18,595	26,630	3
Sling	38,018	15,489	13,311	6
Twitter4j	4,650	1,875	2,742	0
SUM	**1,064,279**	**466,218**	**518,392**	**100**

This balances the different types of errors in our dataset. Afterwards, for each of the 100 faulty methods of the dataset, we created a single unit test that highlights the error in the given method. For each test, we set a timeout of 5 seconds. Finally, we changed the build scripts of the projects to leverage the library and plugin offered by our approach.

All experiments have been conducted on a laptop with a 2.5 GHz Intel CPU and 16 GB of physical memory running macOs 10.13.6.

B. RQ1: Bug Identification

With the first experiment, we want to investigate if our approach is adequate to discover the errors in our dataset. First, using our Assertion Library, one author of the paper manually annotated the classes that contain the errors of our dataset. She annotated the classes with assertions that specify the (assumed) expected properties of the source code. We inserted a single assertion for each error. Then, another author of the paper checked the assertions, verifying that they correctly express the correct behavior of the method in which they were inserted. Second, for each of the 20 projects, we executed the build of the system to create the models and inject the verification framework. Finally, we launched the execution of the 100 tests that we have created. In this manner, we verified the identification of errors at compile time during the build and at runtime during the tests execution. Notice that in this experiment we disabled the recovery strategy.

The execution of the build shows that all 100 errors were correctly detected at compile time. Furthermore, all tests failed due to the exception thrown by our prototype when the assertions are checked at runtime. These results confirm that our implementation is able to identify violations of given specifications.

C. RQ2: Recovery

To answer this research question, we investigated the primary ability of our approach, *i.e.*, to automatically recover the internal state of the program in case of a violation of the specifications. First, we activated the flag in the Assertion Plugin that enables the insertion of the recovery strategy in the final executable. Then, we compiled the projects and ran the tests again so the monitor could identify the errors and we counted how many of them can be successfully recovered. The results show that 84 methods that contain an error of our dataset could successfully pass their tests, resulting in a recovery rate of 84%.

We also investigated the reasons why the recovery strategy failed to recover a proper state in the remaining 16 methods. We first logged the memory state suggested by the recovery strategy. Then, we used it to manually analyze the code and understand the reasons for the failures. We discovered that in 9 cases, the problem was due to the `final` keyword of the Java programming language. Once a field attribute is declared `final`, its value becomes a constant and the Java Virtual Machine (JVM) does not allow to change it. Therefore, the recovery strategy, although it was able to suggest a correct memory state, it was not able to enforce it, due to this restriction of the JVM. In the remaining 7 cases, the over approximation, due to the relaxation of the specification, produced a memory state that did not solve the issue.

With further inspection, we discovered that both types of failures of the recovery strategy could be overcome by changing how the specifications are written. In the case of

```
1  public void poll(long timeout) {
2    long now = time.milliseconds();
3    long deadline = now + timeout;
4    AssertLibrary.ensure(
5      forAll("timeout", and(
6        leq(now, "deadline"), gt("timeout", 1)
7    )));
8    while (now <= deadline) {
9      if(timeout == 1)
10      return;
11     ... //important code which is expected to
12         //be executed at least once
13   }
14 }
```

Listing 4: Example that showcases how to aid the recovery strategy to identify a correct memory state.

TABLE II: Descriptive statistics of the dataset used in our experiments with Source Line Of Code (SLOC), number of statements (STMS), Cyclomatic Complexity (CC), and the size of the generated models (MS).

	SLOC	STMS	CC	MS
Min	43.0	11.0	5.0	10.0
Median	188.0	95.5	43.5	27.5
Average	322.1	167.3	76.9	31.5
Max	1943.0	1170.0	493.0	114.0
Overall	**32206**	**16734**	**7686**	**3148**

`final` attributes, a developer can add an additional assertion before the value of the attribute is set in the constructor. This assertion can verify that the attribute is not assigned with a wrong value, which will later invalidate the other specification. In the case where the recovery strategy does not correctly identify a valid memory state, the specification can be enriched with further details about the expected execution. This tightens the gap between the over approximation and the real correct execution, producing better results.

In the example of Listing 3, the recovery strategy would assign the value 1 to `timeout`, as we have discussed in Section II-C. However, if the same assertion is used with thes code presented in Listing 4, the recovery strategy will not fix the program execution. The memory state with the value 1 for the `timeout` variable is invalid because the execution would stop before reaching the important instructions following the if statement. To solve this problem, the assertion can be enrich with an additional constraint that enforces the variable `timeout` to be greater than 1, as presented by Listing 4 between Line 4 and 7. The assertion can be read as $\forall timeout.(now \leq deadline) \wedge (timeout > 1)$. This specification is stronger than the previous one and it permits the recovery strategy to produce a memory state which enables the correct execution of the method, *e.g.*, `timeout` = 2.

From this experiment, we can conclude that our recovery strategy can successfully recover a correct program execution for 84% of errors encountered. However, we also discovered that its success depends on how the specifications are formulated.

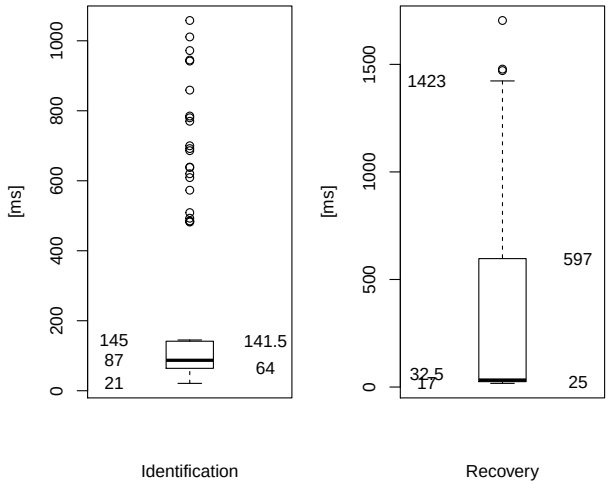

Fig. 7: Boxplot showing the time in ms to identify and recover errors.

D. RQ3: Runtime Overhead

For this research question, we studied the impact of our approach on the runtime performance. We used our dataset composed of 100 errors. Table II shows some descriptive code statistics of the classes that contain the errors of our dataset. We report the statistics per class and not per method because the tests that trigger the errors also perform calls to other methods of the class that contains the faulty method. Moreover, only a single assertion per test is executed. The number of Single Line Of Code (SLOC) varies from 43 to a maximum of 1943, averaging at roughly 322 lines of code per class. The median shows that half of the classes contain around 188 lines of code. The number of statements (STMS), ranges from 11 to 1170, with an average of 167.3 statements and a median of 95.5 statements. The Cyclomatic Complexity (CC) ranges from a minimum of 5 to a maximum of 493 with an average and a median of 76.9 and 43.5, respectively. The size of the models (MS) is computed in terms of the number of declarations in the SMT-Lib notation of the models. The generated models vary from 10 to 114 declarations, while the median and the average are 27.5 and 31.5, respectively.

We ran each of the tests that trigger the 100 errors of our dataset. For each test execution, we computed the time our approach requires to identify the bug (Identification) and the time the recovery strategy requires to produce the memory values (Recovery). We ran each test 5 times and averaged the results. We report the results of our experiments in Figure 7. The time required to identify an error ranges from 21ms to 1058ms, with a median value of 87ms. Furthermore, from the interquartile range of the boxplot we can see that for the majority of the cases, the time required to identify an error ranges between 21ms and 145ms. In contrast, JML [34] requires between 6ms to 215ms to verify the validity of a

single Java statement.

The recovery strategy has a median time lower than the identification of errors, namely 32.5ms. However, it has more variance in the time required to compute the correct values for the memory state, ranging from a 17ms to 1704ms. This result is expected since the recovery strategy proceeds iteratively to weaken the formula. Depending on the number of steps required by the recovery strategy to find the correct memory state, multiple calls to Z3 are performed varying the time required to return the results. To verify our hypothesis, we manually investigated the reasons of this behavior. We profiled our prototype tool and we discovered that 93% of the time is spent in reading and parsing the output of the Z3 process. Whereas for the identification of errors, we read only the exit code of the Z3 process without parsing the output. Furthermore, the maximum number of weakening steps required by the recovery strategy was 2. We also investigated the large number of outliers in the Identification boxplot. We collected the models of these errors and manually verified them with Z3. We discovered that Z3 requires more than half a second to verify them although they have a size between 21 and 40 declarations. The errors discovered with such models are due to overflows and Z3 requires extra time to verify this type of property.

Based on these results, we conclude that our approach for detecting errors at runtime adds a median overhead of 87ms which is comparable to existing approaches. Enabling the recovery strategy adds another median overhead of 32.5ms to the runtime per assertion.

V. Discussion and Threat to Validity

In this section, we discuss the outcome of our evaluation and its implications for researchers and practitioners. Furthermore, we discuss the potential threats to validity of our empirical studies.

A. Summary of the Results

With our three research questions, we studied three aspects of our approach. The first and second research questions investigated the primary purpose of our approach, *i.e.*, to identify and overcome errors in software systems. The third research question studied the runtime overhead imposed by our technique while monitoring the application. The results show that our prototype tool is able to identify all the errors in our dataset and to automatically recover a correct execution behavior for 84% of them, introducing a median runtime overhead of 119.5ms per assertion. We can conclude that our approach is adequate to discover errors in the source code and automatically overcome them when they occur at runtime with an adequate overhead. This has several implications on researchers and developers as discussed in the next subsection.

B. Implications of the Results

Concerning the implications on the research in this area, we presented a general approach that aims at supporting developers to use formal methods for developing software systems. Our approach gives developers the control where to insert and check an assertion. Other approaches, such as JML [1], can follow our methodology so developers can better control and lower the runtime overhead added by the formal verification. More importantly, the results of the second research question show that our recovery strategy can effectively help overcome errors during the program execution. Other approaches can benefit from our rollforward strategy and implement a similar recovery strategy into their toolkit. Moreover, we present our approach for the Java programming language as a case study to show its applicability. Other researchers can adapt our approach to other modern programming languages because it relies on two artifacts: a runtime library and a compiler plugin. Modern programming languages and their toolkits provide both artifacts, making it reasonably easy to adapt our approach. This would enable the possibility to study the effectiveness of our recovery strategy in conjunction with dynamic languages, such as JavaScript, or concurrent languages, such as Rust.

Our results have also several implications for developers. Our approach gives developers the control where to insert and check an assertion. With that, they can decide which statements in the code should be formally verified at compile and runtime. Such flexibility is currently not supported by existing approaches, such as JML [1] and ESC/Java2 [25], where all code statements are always verified. In addition, the results of the first research question show that our approach, given the right assertion, is able to identify *all* errors. The results of the second research question show that our approach allows developers to ship software systems that can automatically recover from errors at runtime. Although we currently support only few types of errors, we showed that with our recovery strategies, users will not encounter failures while developers are notified with the error information and the counter example generated by our approach. Developers can use this information to fix the problems in the next release cycle. While our results show the potential of our approach, more types of errors need to be supported in the future. Furthermore, the results of the third research question show that the runtime overhead added by our approach is acceptable compared to existing approaches, such as [34]–[36].

C. Threats to Validity

In the following section, we discuss threats to the internal and external validity of our evaluation, and how we addressed them in our experiments.

Internal Validity. One threat to the internal validity concerns the reliability of the prototype tool. We mitigated this threat with manual and unit tests. Furthermore, we designed the first research question to investigate the correctness of our implementation with a dataset that consists of real world errors. The application of our approach to the dataset managed to discover all errors, mitigating this possible threat. Another threat to the validity of our study is that we have manually created the assertions used to identify, and later overcome, the errors in the source code of the projects. We mitigate this thread by having two different authors of the paper writing

and then reviewing the assertions. The assertions should be representative of the invariants of the code since all the errors were discovered and only in few instances the recovery strategy was not able to overcome them. Furthermore, this shows that also with a naive knowledge of the system, a simple assertion is adequate to discover and recover errors in the systems. Further studies will be devoted in mining the test cases of the project and use invariant detector tools, *e.g.*, Daikon [37], to automatically infer the assertions and mitigate this threat. Moreover, future work will also leverage failed test case execution to generate further assertions.

With the third research question, we investigated the runtime overhead introduced by our approach. We computed the time required to execute the verification framework inserted into the system. We used the tests that trigger the errors in our dataset instead of running the final artifact to compute its execution time. We decided to rely on the testing phase of the build system because it requires no extra configuration, making the execution of our studies repeatable by other researchers. Furthermore, we computed the timing only from the beginning of the verification of the assertion until the recovery strategy is applied. Therefore, the overhead introduced by the build system does not affect our results. Moreover, background tasks and the scheduler of the operating system could impact the validity of our performance results. We mitigated this threat by repeating the studies five times and averaging the results.

A further threat to the validity of our studies is the missing comparison with other similar techniques that provide a state recovery mechanism, such as ErrDoc [38]. Since C does not have an out-of-the-box mechanism to handle errors like Java, *i.e.*, with Exception, developers are forced to create their own handling error conventions. ErrDoc targets C programs and it fixes errors in error handling code. Furthermore, ErrDoc relies on a specification of the error to generate the fix. The error specification is automatically inferred using APEx [39], while our approach currently requires developers to manually annotate their code. Therefore, a direct comparison between the two approaches is not yet possible.

External Validity. A threat to external validity concerns the generalization of our results in two dimensions: (i) the application of our approach to other software projects and (ii) the extendibility of our approach to other languages and/or build tools. We tested our approach using a controlled environment in which we verified the code using assertions described in FOL formulae with only a single hardware configuration. More engineering efforts will be devoted to extend the expressiveness of the Assertion Library allowing developers to declare properties in other logics. This requires the Assertion Plugin to also generate a more complex model of the source code. Therefore, the generalization of the runtime overhead might not hold for different hardware or projects that require the verification of more complex properties. Moreover, we studied the approach with a limited dataset that comprises 100 bugs collected from 20 open source Java projects. To further extend the dataset or even apply our prototype tool to a full project, we need to provide assertions which specify the correct behavior of the system under analysis. This task is, unfortunately, time consuming and requires expertise in that system. Moreover, we only support the verification of integer and time related properties. Future work will be devoted to extend the datatypes and domains supported by our prototype tool. Furthermore, we plan to test our prototype tool with a real system collaborating with the developers to insert the required assertions.

Other researchers can freely use our tool and apply it to other case studies extending our results. For the extendibility to other languages and/or build tools, our approach relies on two elements: a runtime library and a compiler plugin. Modern programming languages and their toolkits provide both elements, making it reasonably easy to adapt our approach to them.

VI. Related Work

A wide spectrum of related work in literature addresses the verification and repair of software, both at compile- and at run-time. This leads us to divide the related work into three categories: program repair, formal verification at compile time, monitoring and run time enforcement (RV&E).

Program Repair. In the domain of program repair, one of the most used technique to address this problem is a generate-and-validate approach. These techniques monitor the results of the test suites of a project and when a test fails, the repairing technique generates the set of all possible changes, called search space, that can be applied to the code to repair the identified defect. Then, a patch is selected and applied and the failed test is executed again. Based on the outcome of the test, the patch is accepted or refused. Long *et al.* [40] studied the relationship between search space and different existing techniques. They studied how changing the search space, the existing techniques change their repairing success rates. They discovered that for a better correction ration, information outside of the test suites should be included in the search space. Instead, Le *et al.* [41] propose a genetic algorithm to repair defects that can be run in a cloud environment for an average cost of 8$ per patch. Nguyen *et al.* [42] propose an approach that translates the source code into constraint problems based on the tests execution. Then, for each failed test, it generates a constraint and select as repairing patch the piece of code that satisfies such constraint problem. The common shortcoming of these approaches is that they overfit the problem. They generate patches that bypass the program logic to successfully execute the failed tests more than repairing the code. To overcome this, some researchers started to use semantic information to repair the programs. Ke *et al.* [43] proposed to create a corpus of STM problems generated from snippets of code. When a test fails, they derive the input-output relationship of the test and use this to query the corpus. The results are synthesized into patches that are used to repair the program. The work of Mechtaev *et al.* [44] and Tonder and Le Goues [45] are the closest to our approach. Mechtaev *et al.* [44] synthesize patches from a formal specification of the requirements of the project. Tonder and Le Goues [45] model a

program via an intermediate language that performs operation on a heap. Then, developers can write the specifications of their program as heap properties and whenever a property does not hold, their approach synthesizes a patch using the snippets of code where the property holds. All the presented techniques work at compile time and they repair the program when a test fails. In contrast, our approach repair the program at runtime and it does not rely on test to discover errors, therefore it is suitable to also repair failures that were not expected by the developers. Other works follows a similar approach to ours repairing the memory state at runtime. Demsky and Rinard [46] proposed a specification language that is used to express the correctness properties on the data-structures used in the program. These properties are then used at runtime to identify broken data-structures and repair them. Similarly, Lewis and Whitehead [47] proposed a fault-monitor that learns execution invariants. When an invariant is violated, the runtime monitor tries to restore it. In contrast, our approach relies on invariants written directly in the code by the developers in regular Java.

Formal Verification at Compile Time. Various formal verification techniques have been employed to discover software bugs at compile time, among which software model checking deserves a special mention. The effectiveness of these techniques highly depends on the application domain. SPIN [48] is designed to verify protocols, Bandera [49] works on Java programs, SLAM [50] model checks device drivers, and UPPAAL [51] addresses real-time systems. Spalazzi *et al.* [7] propose a verification technique for the correctness of real-time Java software based on a network of timed automata. However, these approaches and tools are monolithic and do not allow user-defined state representation hindering their utilization for different kinds of application domain. Java Pathfinder [52] addresses this problem by providing an open framework that can be customized under the needs of the developers. It employs a modified JVM that is able to interleave symbolic and concrete executions of Java bytecode. Since the verification of program properties is performed on a model, the model must also describe the environment in which the program operates. The implication of this limitation is important in practice because model checking techniques are doomed to perform poorly whenever imprecise models of the environment are provided. Furthermore, such techniques have several scalability issues, *e.g.*, the well-known state-explosion problem and the difficulty in tracking the correctness specifications when software evolves. A different approach is represented by the proof carrying code (PCC) [53]. In this approach, a developer tries to formally prove that the code component, in the assumed initial conditions, fulfills a given safety policy. If such proof can be produced, the code is deployed onto a runtime environment that is able to check (i) whether the proof is a valid proof for the associated code, and (ii) whether the proof implies the desired safety policy, given the initial conditions. In case of a positive answer, the code can be executed on the recipient. The main difference between our approach and PCC is that, even if they both share the idea of shipping the executable with some additional artifacts that enforce the desired property, PCC assumes that the additional artifact is checked only once and statically. This implies that PCC shares the severe scalability issues of other static analysis techniques that limit its applicability to real-world software projects.

Monitoring and RV&E. The limits of formal verification at compile time are overcome by the approach based on monitoring and enforcement at runtime, at the cost of an execution overhead. These are techniques to make safer software systems at run time,*e.g.*, [54]. However, it should be pointed out that, even if on the one hand these techniques overcome the problems of scalability and modeling of the execution environment typical of static analysis techniques; on the other hand, since these techniques are applied at run time, they are necessarily incomplete. Runtime enforcement uses a monitor to detect if the system violates a correctness specification. Subsequently, in case of a violation, it applies counter-actions that bring back the system to a correct state. From this point of view, therefore, runtime verification can be considered a special case of runtime enforcement in which the only counter-action consists in interrupting the execution of the system. There are several works, even recent ones, that show possible applications of this approach [55]. For example, Khan *et al.* [21] present a design methodology for the development of reliable and safe industrial controllers. From the formal specification of an industrial controller executable, it generates the respective program and a monitor that protects the program runtime generated by external attacks. Similarly, Barnett and Schulte [6] have proposed a method to implement the behavioral interface specifications on the .NET platform. In their framework, the models of a program are expressed in AsmL, a specific executable that describes the program's semantics and verifies the monitoring of execution. Colombo and Pace [56], working on a system of financial transactions, integrated static analysis techniques to runtime monitoring techniques. AspectJ [57] and JavaMOP [58] are some of the most used tools for monitoring and applying properties at run time for Java applications. Both allow to intercept events along Java code execution (also called *pointcuts*) and to monitor or react to them. Events that can be captured are object instances, method calls, or variable reads and writes. However, they do not allow attaching an event to a specific line of code. This motivated us to develop the assertion library to be invoked by the programmer exactly in the line of code where control is needed. Furthermore, we argue that, unlike AspectJ and JavaMOP, our approach can be effectively integrated with pre-existing assertion-based test suites. The assertions specified with our assertion library can be used for test generation as well as for static analysis and monitoring. Larva [24] is a tool for generating Java monitors. The main difference to JavaMOP is the more expressive specification language regarding the properties that parameters and system states need to satisfy. Java Modeling Language (JML), proposed by Leavens *et al.* [1], it is perhaps the related work closest to ours. JML supports quantifiers, specification-only variables, and other enhancements with respect to some of its predecessors such

as Eiffel [59]. It compiles a Java program including invariants, pre-, and post-conditions specified by developers. Each source code instruction is wrapped with a set of instructions that guarantees the properties defined by the users. However, this assumes some predefined semantics for the variables used in the program that might be wrong. Furthermore, JML does not provide any kind of automatic recovery of a correct execution behavior.

From a formal point of view, it has been proven that, when RV&E is limited to finite (but arbitrarily long) system runs, any LTL specification can be monitored, but only response properties can be enforced (i.e. properties that can be written as $\Box\Diamond p$ or $\Box(p \rightarrow \Diamond q)$, where p and q are atomic propositions) [60]. Currently, we implemented a prototype where only a subset of the safety specifications are enforced (e.g. those expressible as $\Box p$, where p is a local assertion on program variables). Nevertheless, the theory ensures that our prototype can be extended to the full class of LTL specifications for monitoring and response properties for enforcing.

VII. Conclusion

In this paper, we presented an approach to identify and automatically recover errors in programs. We presented a general method that requires an Assertion Library and an Assertion Plugin. The library is used by developers to assert properties of their code that are verified either at runtime or at compile time by our proposed verification framework. The Assertion Plugin extends the build system generating an abstract behavioral model of the source code and inserts it into the final artifact produced. Furthermore, the Assertion Plugin injects into the final artifact a verification framework. The purpose of the verification framework is twofold: (i) monitor the application runtime and identify inconsistencies and (ii) recover the memory state in order to overcome the identified errors.

We evaluated our approach on a dataset that comprises 100 real world bugs from 20 open source Java projects. Our results show that our approach is able to identify 100% of the errors and automatically recover from 84% of them. Future work will be devoted to enrich the logic theories supported by our prototype and perform case studies with developers to fully integrate and study our approach in complex systems.

Acknowledgment

This research is funded by the Austrian Research Promotion Agency FFG within the FFG Bridge 1 program, grant no. 850757.

References

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[2] R. Calinescu, K. Johnson, and C. Paterson, "Efficient parametric model checking using domain-specific modelling patterns," in *Proceedings of the 40th International Conference on Software Engineering (ICSE): New Ideas and Emerging Results*. ACM, 2018, pp. 61–64.

[3] N. Tillmann and J. De Halleux, "PEX–white box test generation for .NET," in *International Conference on Tests and Proofs (TAP)*. Springer, 2008, pp. 134–153.

[4] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[5] W. Grieskamp, N. Tillmann, and W. Schulte, "XRT exploring runtime for .NET architecture and applications," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 3, pp. 3–26, 2006.

[6] M. Barnett and W. Schulte, "Runtime verification of .NET contracts," *Journal of Systems and Software*, vol. 65, no. 3, pp. 199–208, 2003.

[7] L. Spalazzi, F. Spegni, G. Liva, and M. Pinzger, "Towards model checking security of real time Java software," in *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 642–649.

[8] G. Liva, M. T. Khan, F. Spegni, L. Spalazzi, A. Bollin, and M. Pinzger, "Modeling time in Java programs for automatic error detection," in *Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE)*, 2018, pp. 50–59.

[9] G. Liva, M. T. Khan, and M. Pinzger, "Extracting timed automata from Java methods," in *Proceedings of the IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017, pp. 91–100.

[10] P. Ferrara, A. Cortesi, and F. Spoto, "CIL to Java-bytecode translation for static analysis leveraging," in *Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE)*. ACM, 2018, pp. 40–49.

[11] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix-points," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[13] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[14] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, 2008.

[15] Facebook, "Infer static analyzer," 2017, accessed 10 May 2019. [Online]. Available: http://fbinfer.com/

[16] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, "Fast and precise type checking for JavaScript," *Proceedings of the ACM on Programming Languages*, vol. 1, no. 48, pp. 1–30, 2017.

[17] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 580–586.

[18] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[19] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[20] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, Inc., 2017.

[21] M. T. Khan, D. Serpanos, and H. Shrobe, "ARMET: Behavior-based secure and resilient industrial control systems," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 129–143, 2018.

[22] F. Chen, M. d'Amorim, and G. Roşu, *A formal monitoring-based framework for software development and analysis (ICFEM)*, 2004.

[23] S. Pinisetty, G. Schneider, and D. Sands, "Runtime verification of hyperproperties for deterministic programs," in *Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE)*. ACM, 2018, pp. 20–29.

[24] C. Colombo, G. J. Pace, and G. Schneider, "LARVA—safer monitoring of real-time Java programs," in *Proceedings of the 7th International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 2009, pp. 33–37.

[25] D. R. Cok and J. R. Kiniry, "Esc/java2: Uniting esc/java and jml," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 2004, pp. 108–128.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[27] R. R. Smullyan, *First-order logic*. Springer Science & Business Media, 2012, vol. 43.

[28] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[29] D. Gabbay, "The declarative past and imperative future," in *Temporal logic in specification*. Springer, 1989, pp. 409–448.

[30] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[31] P. Godefroid, "Compositional dynamic test generation," in *ACM Sigplan Notices*, vol. 42, no. 1. ACM, 2007, pp. 47–54.

[32] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[33] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[34] E. Rodríguez, M. B. Dwyer, J. Hatcliff *et al.*, "Checking jml specifications using an extensible software model checking framework," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 280–299, 2006.

[35] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 782–791.

[36] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *Acm Sigplan Notices*, vol. 42, no. 10. ACM, 2007, pp. 569–588.

[37] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[38] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 752–762.

[39] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 472–482.

[40] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 702–713.

[41] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[42] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 772–781.

[43] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 295–306.

[44] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 129–139.

[45] R. Van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 151–162.

[46] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Acm sigplan notices*, vol. 38, no. 11. ACM, 2003, pp. 78–95.

[47] C. Lewis and J. Whitehead, "Runtime repair of software faults using event-driven monitoring," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 275–280.

[48] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[49] J. Hatcliff and M. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," in *International Conference on Concurrency Theory (CONCUR)*. Springer, 2001, pp. 39–58.

[50] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 1–3.

[51] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[52] G. Brat, K. Havelund, S. Park, and W. Visser, "Java PathFinder - second generation of a Java model checker," in *In Proceedings of the Workshop on Advances in Verification*, 2000.

[53] G. C. Necula, "Proof-carrying code," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119, 1997.

[54] Y. Falcone, K. Havelund, and G. Reger, *A tutorial on runtime verification*. IOS Press, 2013, vol. 34.

[55] E. Bartocci and Y. Falcone, *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, 2018, vol. 10457.

[56] C. Colombo and G. J. Pace, "Industrial experiences with runtime verification of financial transaction systems: lessons learnt and standing challenges," in *Lectures on Runtime Verification*. Springer, 2018, pp. 211–232.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2001, pp. 327–354.

[58] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2005, pp. 546–550.

[59] B. Meyer, "Eiffel*: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.

[60] Y. Falcone, J.-C. Fernandez, and L. Mounier, "What can you verify and enforce at runtime?" *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 349–382, 2012.

AAU-SERG-2019-002
ISSN 1872-5392