RV'16, Lecture Notes in Computer Science 10012, pp 46-67. 2016

Runtime Verification at Work: A Tutorial

Philip Daian, Dwight Guth, Chris Hathhorn, Yilong Li, Edgar Pek, Manasvi Saxena, Traian Florin Șerbănuță, and Grigore Roșu

Runtime Verification Inc., University of Illinois at Urbana-Champaign

Abstract. We present a suite of runtime verification tools developed by Runtime Verification Inc.: RV-Match, RV-Predict, and RV-Monitor. RV-Match is a tool for checking C programs for undefined behavior and other common programmer mistakes. It is extracted from the most complete formal semantics of the C11 language and beats many similar tools in its ability to catch a broad range of undesirable behaviors. RV-Predict is a dynamic data race detector for Java and C/C++ programs. It is perhaps the only tool that is both sound and maximal: it only reports real races and it can find all races that can be found by any other sound data race detector analyzing the same execution trace. RV-Monitor is a runtime monitoring tool that checks and enforces safety and security properties during program execution. Our tools focus on reporting no false positives and are free for non-commercial use.

1 Introduction

Runtime verification is an analysis and execution approach based on extracting information from a running system and using it to detect, and possibly react to, observed behaviors satisfying or violating certain properties. In this session, we present the practical applications of runtime verification technology that we are currently exploring.

Runtime verification avoids the complexity of traditional formal verification techniques (like model checking and theorem proving) by analyzing only one or a few execution traces and working directly with the actual system. Thus, runtime verification scales up relatively well and gives more confidence in the results of the analysis because it avoids the tedious and error-prone step of formally modeling the system (at the expense of reduced coverage). Moreover, through its reflective capabilities, runtime verification can be made an integral part of the target system, monitoring and guiding its execution during deployment.

We present three instantiations of the runtime verification approach: (1) RV-Match, a dynamic analysis tool for finding a wide range of flaws in C programs, (2) RV-Predict, a dynamic data race detector for Java and C, and (3) RV-Monitor, a runtime monitoring framework for checking and enforcing properties of Java and C programs.[1]

[1] See `https://runtimeverification.com/` for an overview of our tools and company.

2 RV-Match

RV-Match is a tool for checking C programs for undefined behavior and other common programmer mistakes. It is extracted from the most complete formal semantics of the C11 language. Previous versions of this tool were used primarily for testing the correctness of the semantics, but we have improved it into a tool for doing practical analysis of real C programs. It beats many similar tools in its ability to catch a broad range of undesirable behaviors. We demonstrate this below with comparisons based on a third-party benchmark.

2.1 Background: RV-Match

The $\mathbb{K}$ semantic framework[2] is a program analysis environment based on term rewriting [1]. Users define the formal semantics of a target programming language and the $\mathbb{K}$ framework provides a series of formal analysis tools specialized for that language, such as a symbolic execution engine, a semantic debugger, a systematic checker for undesired behaviors (model checker), and even a fully-fledged deductive program verifier. Our tool, RV-Match, is based on the $\mathbb{K}$ framework instantiated with the publicly-available C11 semantics[3] [8, 9], a rigorous formalization of the current ISO C11 standard [15]. We have specially optimized RV-Match for the execution and detection of errors in C programs.

Unlike modern optimizing compilers, which have a goal to produce binaries that are as small and as fast as possible at the expense of compiling programs that may be semantically incorrect, RV-Match instead aims at mathematically rigorous dynamic checking of programs for strict conformance with the ISO C11 standard. A strictly-conforming program is one that does not rely on implementation-specific behaviors and is free of the most notorious feature of the C language, *undefined behavior*. Undefined behaviors are semantic holes left by the standard for implementations to fill in. They are the source of many subtle bugs and security issues [13].

2.2 Running RV-Match

Users interface with RV-Match through the `rv-match` executable, which behaves as a drop-in replacement for compilers like `gcc` and `clang`. Consider a file `undef.c` with contents:

```
int main(void) {
    int a;
    &a+2;
}
```

We compile the program with `rv-match` just as we would with `gcc` or `clang`. This produces an executable named `a.out` by default, which should behave just

[2] See `http://kframework.org` for details.
[3] Available at `https://github.com/kframework/c-semantics`.

as an executable produced by another compiler—for strictly-conforming, valid programs. For undefined or invalid programs, however, `rv-match` reports errors and exits if it cannot recover:

```
$ rv-match undef.c
$ ./a.out
Error: UB-CEA1
Description: A pointer (or array subscript) outside the
  bounds of an object.
Type: Undefined behavior.
See also: C11 sec. 6.5.6:8, J.2:1 item 46
  at main(undef.c:2)
```

In addition to location information and a stack trace, RV-Match also cites relevant sections of the standard [15].

2.3 Finding undefined behavior in C using RV-Match

Below, we describe several examples demonstrating RV-Match's capabilities for detecting undefined behavior. Note that these examples cover only a small subset of the errors which RV-Match detects.

Unsequenced side effects. Consider a simple program:

```
int main(void) {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

Compiled with `clang`, this program returns 3. With `gcc`, however, it returns 4, because `gcc` chooses to sequence both assignments before the addition expression. In general, compilers are allowed to introduce optimizations as long as they do not affect the behavior of well-defined programs. However, since this program is undefined, such optimizations can have unexpected consequences. When compiled with `rv-match`, we get the following output after running the program:

```
Error: UB-EIO8
Description: Unsequenced side effect on scalar object
  with side effect of same object.
Type: Undefined behavior.
See also: C11 sec. 6.5:2, J.2:1 item 35
  at main(1-unsequenced-side-effect.c:3)
```

Buffer overflows. Perhaps the most notorious errors in C programs are buffer overflows. RV-Match is capable of detecting all varieties of buffer overflows.[4]

[4] See `2-buffer-overflow.c` from the `examples/demo` directory of the `c-semantics` repository at `https://github.com/kframework/c-semantics` for examples of several varieties.

As a more subtle example, consider an overflow within the subobjects of an aggregate type (in this case, a `struct` of an array followed by an integer):

```
struct foo { char buffer[32]; int secret; };
int idx = 0;
void setIdx() { idx = 32; }
int main(void) {
    setIdx();
    struct foo x = {0};
    x.secret = 5;
    return x.buffer[idx];
}
```

We can safely assume the `struct` is laid out sequentially in memory, yet access to the 32nd index of this array is still undefined behavior. Tools like `valgrind` usually do not catch this sort of issue because the accesses will be to valid addresses for other pieces of the aggregate. But it is still undefined behavior. `gcc` compiles the program and execution leads to a leak of the secret integer. RV-Match, however, will detect the flaw:

```
Error: UB-CER4
Description: Dereferencing a pointer past the end of an
  array.
Type: Undefined behavior.
See also: C11 sec. 6.5.6:8, J.2:1 item 47
  at main(3-array-in-struct.c:16)
```

RV-Match reports more than 150 varieties of error, like UB-CER4 above, most of which concern undefined behavior.[5]

Implementation-defined behavior. RV-Match is also able to detect errors related to implementation-defined behavior, which the C standard defines as unspecified behavior where each implementation documents how the choice is made. `rv-match` can be instantiated with different profiles corresponding to different implementation choices (use `rv-match -v` for existing choices). An example of implementation-defined behavior is the conversion to a type that cannot store a specified value, thus triggering a loss of precision.

More examples of undefined behavior and the features of RV-Match are described in the "Running Examples" section of the RV-Match documentation [11].

2.4 Evaluation

Of course, there is no shortage of tools for analyzing C programs. To evaluate the strengths of our tool, we compare RV-Match against some popular C analyzers

[5] For a list of the errors and example programs demonstrating them, see `https://github.com/kframework/c-semantics/blob/master/examples/error-codes`.

on a benchmark from Toyota ITC. We also briefly mention our experience with running our tool on the SV-COMP benchmark. The other tools we consider are listed below:

- *GrammaTech CodeSonar* is a static analysis tool for identifying "bugs that can result in system crashes, unexpected behavior, and security breaches" [10].
- *MathWorks Polyspace Bug Finder* is a static analyzer for identifying "run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software" [22].
- *MathWorks Polyspace Code Prover* is a tool based on abstract interpretation that "proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other runtime errors in C and C++ source code" [23].
- *Clang UBSan, TSan, MSan, and ASan (version 3.7.1)* are all `clang` modules for instrumenting compiled binaries with various mechanisms for detecting undefined behavior, data races, uninitialized reads, and various memory issues, respectively [6].
- *Valgrind Memcheck and Helgrind (version 3.10.1, GCC version 4.8.4)* are tools for instrumenting binaries for the detection of several memory and thread-related issues (illegal reads/writes, use of uninitialized or unaddressable values, deadlocks, data races, etc.) [24].
- *The CompCert C interpreter (version 2.6)* uses an approach similar to our own. It executes programs according to the semantics used by the CompCert compiler [4] and reports undefined behavior.
- *Frama-C Value Analysis (version sodium-20150201)*, like Code Prover, is a tool based on static analysis and abstract interpretation for catching several forms of undefinedness [5].

The Toyota ITC benchmark [25]. This publicly-available[6] benchmark consists of 1,276 tests, half with planted defects meant to evaluate the defect rate capability of analysis tools and the other half without defects meant to evaluate the false positive rate. The tests are grouped in nine categories: static memory, dynamic memory, stack-related, numerical, resource management, pointer-related, concurrency, inappropriate code, and miscellaneous.

We evaluated RV-Match along with the tools mentioned above on this benchmark. Our results appear in Figure 1 and the tools we used for our evaluation are available online.[7] Following the method of Shiraishi, Mohan, and Marimuthu [25], we report the value of three metrics: DR is the detection rate, the percentage of tests containing errors where the error was detected; $\underline{\mathsf{FPR}} = 100 - \mathsf{FPR}$, where FPR is the false positive rate; and PM is a productivity metric, where $\mathsf{PM} = \sqrt{\mathsf{DR} \times \underline{\mathsf{FPR}}}$, the geometric mean of DR and $\underline{\mathsf{FPR}}$.

Interestingly, and similar to our experience with the SV-COMP benchmark mentioned below, the use of RV-Match on the Toyota ITC benchmark detected a number of flaws in the benchmark itself, both in the form of undefined behavior

[6] See `https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark`.

[7] For tools and instructions on reproducing these results, see `https://github.com/runtimeverification/evaluation/tree/master/toyota-itc-benchmark`.

that was not intended, and in the form of tests that were intended to contain a defect but were actually correct. Our fixes for these issues were accepted by the Toyota ITC authors and we used the fixed version of the benchmark in our experiments. Unfortunately, we do not have access to the MathWorks and GrammaTech static analysis tools, so in Figure 1 we have reproduced the results reported in [25]. Thus, it is possible that the metrics scored for the other tools may be off by some amount.

The SV-COMP benchmark suite. This consist of a large number of C programs used as verification tasks during the International Competition on Software Verification (SV-COMP) [3]. We analyzed 1346 programs classified as correct with RV-Match and observed that 188 (14%) of the programs exhibited undefined behavior. Issues ranged from using uninitialized values in expressions, potentially invalid conversions, incompatible declarations, to more subtle strict aliasing violations. Our detailed results are available online.[8]

3 RV-Predict

RV-Predict is a dynamic data race detector for Java and C/C++ programs. RV-Predict is perhaps the only tool that is both sound and maximal: it only reports real races and it can find all races that can be found by any other sound data race detector analyzing the same execution trace. We have evaluated our tool on a set of real Java programs and we have been able to find a large number of previously unknown data race violations. We report a case study on testing Tomcat, a widely used Java application server, using RV-Predict. Moreover, we have obtained encouraging results after evaluating RV-Predict on a smaller class of C programs. The more mature Java version can perform both online (i.e., data races detection as the program runs) and offline (i.e. data race detection after the program is run and traces are collected) analysis. The C/C++ version is a prototype and can only perform offline data race detection.

3.1 Background: RV-Predict

RV-Predict is based on an important theoretical result by Șerbănuță, Chen, and Roșu [2]: given an execution trace, it is possible to build a *maximal and sound* causal model for concurrent computations. This model has the property that it consists of all traces that any program capable of generating the original trace can also generate.

Moreover, based on the results of Huang, Meredith, and Roșu [14], RV-Predict implements a technique that provides a provably higher detection capability than the state-of-the-art techniques. A crucial insight behind this technique is the inclusion of abstracted control flow information in the execution model, which expands the space of the causal model provided by classical

[8] Detailed SV-COMP benchmark results are available at `https://github.com/runtimeverification/evaluation/tree/master/svcomp-benchmark`.

RV'16, Lecture Notes in Computer Science 10012, pp 46-67. 2016

Tool		Static memory	Dynamic memory	Stack-related	Numerical	Resource management	Pointer-related	Concurrency	Inappropriate code	Misc.	Avg. (unweighted)	Avg. (weighted)
RV-Match (`rv-match`)	DR	100	94	100	96	93	98	67	*0*	63	79	82
	FPR	100	100	100	100	100	100	100	–	100	100	100
	PM	100	97	100	98	96	99	82	*0*	79	89	91
GrammaTech CodeSonar	DR	100	89	*0*	48	61	52	70	46	69	59	68
	FPR	100	100	–	100	100	96	77	99	100	97	98
	PM	100	94	*0*	69	78	71	73	67	83	76	82
MathWorks Bug Finder	DR	97	90	15	41	55	69	*0*	28	69	52	62
	FPR	100	100	85	100	100	100	–	94	100	98	99
	PM	98	95	36	64	74	83	*0*	51	83	71	78
MathWorks Code Prover	DR	97	92	60	55	20	69	*0*	1	83	53	53
	FPR	100	95	70	99	90	93	–	97	100	94	95
	PM	98	93	65	74	42	80	*0*	10	91	71	71
UBSan + TSan + MSan + ASan (`clang`)	DR	79	16	95	59	47	58	67	*0*	37	51	47
	FPR	100	95	75	100	96	97	72	–	100	93	95
	PM	89	39	84	77	67	75	70	*0*	61	69	67
Valgrind + Helgrind (`gcc`)	DR	9	80	70	22	57	60	72	2	29	44	42
	FPR	100	95	80	100	100	100	79	100	100	95	97
	PM	30	87	75	47	76	77	76	13	53	65	65
CompCert interpreter	DR	97	29	35	48	32	87	58	17	63	52	51
	FPR	82	80	70	79	83	73	42	83	71	74	76
	PM	89	48	49	62	52	80	49	38	67	62	63
Frama-C Value Analysis	DR	82	79	45	79	63	81	7	33	83	61	66
	FPR	96	27	65	47	46	40	100	63	49	59	55
	PM	89	46	54	61	54	57	26	45	63	60	60

Fig. 1. Comparison of tools on the 1,276 tests of the ITC benchmark. The numbers for the GrammaTech and MathWorks tools come from Shiraishi, Mohan, and Marimuthu [25].

- Blue indicates the best score in a category for a particular metric, while orange emphasizes the weighted average of the productivity metric for each tool.
- DR, FPR, and PM are, respectively, the detection rate, 100 − FPR (the complement of the false positive rate), and the productivity metric.
- The final average is weighted by the number of tests in each category.
- Italics and a dash indicate categories for which a tool has no support.

happens-before or causally-precedes race detection techniques. We encode the control flow and a minimal set of feasibility constraints in first-order logic, thus reducing the race detection problem to a constraint satisfaction problem, which can be efficiently solved by SMT solvers.

3.2 Running RV-Predict

The Java version. RV-Predict can be run both from the command line, as a drop-in replacement for the `java` command, and as an agent, to ease integration with IDEs and build management tools like Maven. For more details, please refer to the "Running RV-Predict" section of the documentation [19].

The C/C++ version. Running RV-Predict for C/C++ involves two steps. In the first step, we create an instrumented version of the multithreaded C/C++ program. In the second step, RV-Predict's backend performs offline data race predictive analysis based on the principles described in the background section. Concretely, given a C program `file.c`, the two steps are shown below:

```
$ rv-predict-c-compile file.c
$ rv-predict-execute ./a.out
```

3.3 Detecting common data-race patterns using RV-Predict

Data races are common concurrency bugs in multithreaded programs. A data race can be defined as two threads accessing a shared memory location concurrently and at least one of the accesses is a write. Data races are notoriously difficult to find and reproduce because they often happen under very specific circumstances. They usually manifest as intermittent or non-deterministic failures during testing. And when failures do occur, they typically produce mysterious error messages, far from the root cause of the data race.

Despite all of the work on solving this problem, it remains a challenge in practice to detect data races effectively and efficiently. In this section, we summarize common classes of data races and show how to detect them with RV-Predict. The examples described below can be found in the RV-Predict distribution.

A simple data race. The simplest data race is also the most frequent in practice: two threads access a shared variable without synchronization. In Java, a shared variable is either a field (instance or static) or an array element. See JLS section 17.4.1 for the precise definition. For example:

```
public class SimpleRace {
    static int sharedVar;
    public static void main(String[] args) {
        new ThreadRunner() {
            @Override public void thread1() {sharedVar++;}
            @Override public void thread2() {sharedVar++;}
        };
    }
}
```

The access to `sharedVar` is not synchronized. Note that the `ThreadRunner` class (see the simple race description on our blog [18]) is a utility class containing boilerplate code that instantiates two threads with the defined tasks (it will be used throughout this section to simplify descriptions).

RV-Predict detects this race and the report it generates is below. For readability, the code above only shows the core of the problem, and so the lines in the report do not match line numbers in the code above.

```
Data race on field examples.SimpleRace.sharedVar: {{{
    Concurrent write in thread T10 (locks held: {})
 ---->  at examples.SimpleRace$1.thread1(SimpleRace.java:11)
        at examples.ThreadRunner$1.run(ThreadRunner.java:17)
    T10 is created by T1
        at examples.ThreadRunner.<init>(ThreadRunner.java:26)

    Concurrent read in thread T11 (locks held: {})
 ---->  at examples.SimpleRace$1.thread2(SimpleRace.java:16)
        at examples.ThreadRunner$2.run(ThreadRunner.java:23)
    T11 is created by T1
        at examples.ThreadRunner.<init>(ThreadRunner.java:27)
}}}
```

Although this particular data race might be easy to spot through code review, similar instances buried deep in thousands of lines of code can be very hard to discover. As shown above, RV-Predict can make detection of such bugs simple because it provides the precise location where conflicting memory accesses occur, the stack traces of the two threads involved in the race, the point of thread creation, and the locks held by each thread.

Even while developing RV-Predict, we were able to find a variation of this bug in our own RV-Match code base. Specifically, there was an intermittently occurring null pointer exception in the parser implementation. A (still) standard approach for debugging such issues relies on reproducing the bug and attempting to track the behavior backward to the root of the issue. Such an approach can be tedious and time consuming—RV-Predict, by contrast, finds these issues with minimal effort (for the story on our experience of using RV-Predict to debug RV-Match, see our blog [12]).

Using a non-thread-safe class without synchronization. This class of bugs occurs if a developer assumes that the class being used is thread-safe. In fact many classes are not designed to be used in a multithreaded environment, e.g., `java.util.ArrayList`, `java.util.HashMap`, and many other classes in the Java Collections Framework, so unwarranted thread-safety assumptions can easily creep into the code. Consider this example:

```
import java.util.ArrayList;
import java.util.List;
public class RaceOnArrayList {
    static List<Integer> list = new ArrayList<>();
    public static void main(String[] args) {
        new ThreadRunner() {
LX:         @Override public void thread1() {list.add(0);}
LY:         @Override public void thread2() {list.add(1);}
        };
    }
}
```

Both threads are trying to add an element to the `ArrayList`. However, since the underlying data structure is not thread-safe, and the client code does not perform synchronization, there exists a data race.

Simply running this example will *occasionally* trigger an exception, indicating that something went wrong, but there is no guarantee this bug will appear during testing. Below is the shortened output from RV-Predict (exact values of line numbers are replaced with symbolic values `LX` and `LY`):

```
Data race on field java.util.ArrayList.$state: {{{
    Concurrent read in thread T10 (locks held: {})
 ---->  at examples.RaceOnArrayList$1.thread1(RaceOnArrayList.java:LX)
    ...

    Concurrent write in thread T11 (locks held: {})
 ---->  at examples.RaceOnArrayList$1.thread2(RaceOnArrayList.java:LY)
    ...
}}}
```

Notice that RV-Predict reports the race on the symbolic field `$state`, rather than the field inside the class `ArrayList` where the race actually occurs. This is by design: RV-Predict's error messages abstract away from the low-level implementation details of the Java class library to make identifying the root cause of a data race easier.

A more complex example violates rule LCK04-J from the CERT Oracle Coding Standard for Java: "Do not synchronize on a collection view if the backing collection is accessible." The CERT standard continues:

> Any class that uses a collection view rather than the backing collection as the lock object may end up with two distinct locking strategies. When the backing collection is accessible to multiple threads, the class that locked on the collection view has violated the thread-safety properties and is unsafe. Consequently, programs that both require synchronization while iterating over collection views and have accessible backing collections must synchronize on the backing collection; synchronization on the view is a violation of this rule. [20]

In the example below, `map` is an already synchronized map backed by a `HashMap`. When the first thread inserts a key-value pair into the map, the second thread acquires the monitor of `keySet` and iterates over the key set of the map. This is a direct violation of LCK04-J: thread 2 incorrectly synchronizes on `keySet` instead of `map`.

```
public class RaceOnSynchronizedMap {
    static Map<Integer, Integer> map =
        Collections.synchronizedMap(new HashMap<>());
    public static void main(String[] args) {
        new ThreadRunner() {
LX:         @Override public void thread1() {
                map.put(1, 1);
            }
            @Override public void thread2() {
                Set<Integer> keySet = map.keySet();
LY:             synchronized (keySet) {
LY':                for (int k : keySet)
                        System.out.println("key = " + k);
                }
            }
        };
    }
}
```

Thankfully, RV-Predict reports the race condition and reveals the underlying reason—two threads are holding different monitors:

```
Data race on field java.util.HashMap.$state: {{{
    Concurrent write in thread T10 (locks held: {Monitor@722c41f4})
 ---->  at examples.RaceOnSynchronizedMap$1.thread1(
          RaceOnSynchronizedMap.java:LX)
        - locked Monitor@722c41f4 at
        examples.RaceOnSynchronizedMap$1.thread1
        (RaceOnSynchronizedMap.java:LX)
        ...
    Concurrent read in thread T11 (locks held: {Monitor@1f72ae1d})
 ---->  at examples.RaceOnSynchronizedMap$1.thread2(
          RaceOnSynchronizedMap.java:LY')
        - locked Monitor@1f72ae1d at
        examples.RaceOnSynchronizedMap$1.thread2
        (RaceOnSynchronizedMap.java:LY)
        ...
}}}
```

Broken spinning loop. Often, we want to synchronize multiple threads based on some condition. We might achieve this by using a `while` loop to block until the condition becomes satisfied. For example:

```
public class BrokenSpinningLoop {
    static int sharedVar;
    static boolean condition = false;
    public static void main(String[] args) {
        new ThreadRunner() {
            @Override public void thread1() {
                sharedVar = 1;
                condition = true;
            }
            @Override public void thread2() {
                while (!condition) Thread.yield();
                if (sharedVar != 1)
                    throw new RuntimeException(
                        "How is this possible!?");
            }
        };
    }
}
```

How can this program ever throw the `RuntimeException`? The data race on `condition` might be obvious, but it appears to be innocuous. The exception should be impossible regardless of how accesses to `condition` are ordered.

Nonetheless, the exception can, in fact, be raised, because thread 2, after passing the `while` loop, might still read 0 instead of 1 from `sharedVar`. This can be due to several reasons, such as reordering and caching. In fact, the Java memory model allows such counter-intuitive behavior to happen when the program contains any data races at all. In one instance, this type of bug directly caused a loss of $12 million worth of lab equipment [26].

More examples of data race patterns that RV-Predict detects can be found on our website [18].

3.4 The RV-Predict backend: prediction power vs. efficiency

By default, RV-Predict attempts to strike a good balance between efficiency and prediction power. Nevertheless, while the default settings were engineered to work for most common cases, there might be cases where user input could improve the prediction process. We provide several options for advanced users to tune RV-Predict:

1. *Window size.* For efficiency reasons, RV-Predict splits the execution trace into segments (called windows) of a specified size. The default window size is 1000. Users can alter this size using the `--window` option, with the intuition that a larger size provides better coverage at the expense of increasing the analysis time.
2. *Excluding packages.* To allow better control over the efficiency, RV-Predict provides the option `--exclude` to remove certain packages from logging. This option takes a list of package pattern prefixes separated by commas and excludes from logging any class matched by one of the patterns. The

patterns can use `*` to match any sequence of characters. Moreover, `*` is automatically assumed at the end of each pattern (to make sure inner classes are excluded together with their parent). Note that excluding packages might affect precision, as events from non-logged packages might prevent certain race conditions from occurring.

3. *Including packages.* For more flexibility in selecting which packages to include and exclude, RV-Predict also provides the `--include` option, which is similar to the `--exclude` option (it accepts a comma separated list of package patterns), but opposite in effect.

3.5 Running RV-Predict on Tomcat

When developers are dealing with a large project using multiple kinds of synchronization mechanisms, debugging becomes much more difficult and often requires a thorough understanding of the system. But RV-Predict can help with this task regardless of the code size of the project. As an example, we have run RV-Predict on Tomcat, one of the most widely used Java application servers.

Integrating RV-Predict into Tomcat's build cycle is straightforward. It essentially boils down to assuring that the RV-Predict agent is run with the unit tests. The only required change is in `build.xml`:

```
<jvmarg value="-javaagent:${rvPath}/rv-predict.jar=--base-log-dir log" />
```

Where `${rvPath}` is RV-Predict's installation path and `log` is the location where RV-Predict will store its logs and results. RV-Predict runs along with unit-tests and its inclusion introduces a runtime overhead of roughly 5x (i.e., from 50 minutes to 260 minutes). We performed these experiments on Tomcat 8.0.26 and RV-Predict found almost 40 unique data races in only a few runs. All bugs were reported to developers and fixed in the next release.[9]

4 RV-Monitor

RV-Monitor is a software analysis and development framework that aims to reduce the gap between specification and implementation by allowing them together to form a system [21]. With RV-Monitor, runtime monitoring is supported and encouraged as a fundamental principle for building reliable software: monitors are synthesized from specifications and integrated into the original system to check its behavior during execution.

RV-Monitor evolved from the popular JavaMOP runtime verification framework [16] and represents an effort to create a robust, extendable framework for monitoring library generation. In this section, we will show examples of RV-Monitor compiling specifications into code, for both desktop applications and embedded systems.

[9] See `https://goo.gl/LOOhWt` for a list of the bugs and `http://tomcat.apache.org/tomcat-8.0-doc/changelog.html#Tomcat_8.0.27_(markt)` for the Tomcat 8.0.27 changelog.

4.1 Background: RV-Monitor

Monitoring executions of a system against expected properties plays an important role in both the software development process (e.g., during debugging and testing) and as a mechanism for increasing the reliability and security of deployed systems. Monitoring a program execution generates a trace comprising events of interest. When an execution trace validates or violates a property, the monitor triggers actions appropriate to its purpose in the system [16]. RV-Monitor is a parametric monitoring system, i.e., it allows the specifications of properties that relate objects in the program, as well as global properties. Our approach consists of two phases: in the first phase, the execution trace is sliced according to a parameter instance, while in the second phase each slice is checked by a monitor dedicated to the slice.

At its core, RV-Monitor allows users to specify properties that the system should satisfy at runtime (safety or security properties, API protocols, etc.) and then generate efficient monitoring libraries for them. The generated libraries can then be used in two ways, either (1) manually, by calling the monitoring methods at the desired places, or (2) automatically, by inserting calls to the monitoring methods using instrumentation mechanisms.

When a specification is violated or validated during program execution, user-defined actions are triggered. The triggered actions can range from logging to runtime recovery. RV-Monitor can be considered from at least three perspectives: (1) as a discipline allowing one to improve safety, reliability and dependability of a system by monitoring its requirements against its implementation at runtime; (2) as an extension of programming languages with logics (one can add logical statements anywhere in the program, referring to past or future states); and (3) as a lightweight formal method.

RV-Monitor takes as input one or more specification files and generates Java classes that implement the monitoring functionality defined therein. Each RV-Monitor specification defines a number of events, which represent abstractions of certain points in programs, e.g., a call to the `hasNext()` method in Java, or closing a file. With these event abstractions in mind, a user can define one or more properties over the events, taking the events as either atoms in logical formulae or as symbols in formal language descriptions. For example, the user may use these events as symbols in a regular expression or as atoms in a linear temporal logic formula. In the generated Java class, each event becomes a method that can be either called manually by a user or inserted automatically by using some means of instrumentation, such as AspectJ.

Each specification also has a number of handlers associated with each property that are run when the associated property matches some specific conditions. For instance, when a regular expression pattern matches, we run a handler designated by the keyword `@match`, and when a linear temporal logic property is violated, we run a handler designated by the keyword `@violation`. Additionally, RV-Monitor is able to generate monitors that enforce a given property by delaying threads in multithreaded programs.

4.2 Running RV-Monitor

As mentioned above, calls to the event methods generated by RV-Monitor can either be manually added to programs or programs can be automatically instrumented. Note that the examples in this section and the following sections are included as part of the RV-Monitor distribution and available online.[10]

The manual instrumentation method. Manual calls may appear tedious at first, but they allow for fine grain use of RV-Monitor monitors as a programming paradigm. For example, consider the RV-Monitor `HasNext.rvm` property:

```
package rvm;
HasNext(Iterator i) {
    event hasnext(Iterator i) { }
    event next(Iterator i) { }
    ere : (hasnext hasnext* next)*
    @fail {
        System.out.println(
            "! hasNext() has not been called before "
            + "calling next() for an iterator");
            __RESET;
    }
}
```

Now the generated Java monitoring library (named `HasNextRuntimeMonitor` after the property) will contain two methods (one for each event), with the following signatures:

```
public static final void hasNextEvent(Iterator i)
public static final void nextEvent(Iterator i)
```

By calling these methods directly, we can control exactly what we wish to monitor. For instance, we can add a wrapper class for `Iterator` that has versions `hasNext` and `next` that call our monitoring code and only use them in places where correctness is crucial. The class could be defined as follows:

[10] See `https://github.com/runtimeverification/javamop/tree/master/examples`.

```
public class SafeIterator<E>
        implements java.util.Iterator<E> {
    private java.util.Iterator<E> it;
    public SafeIterator(java.util.Iterator it) {
        this.it = it;
    }
    public boolean hasNext() {
        rvm.HasNextRuntimeMonitor.hasnextEvent(it);
        return it.hasNext();
    }
    public E next() {
        rvm.HasNextRuntimeMonitor.nextEvent(it);
        return it.next();
    }
    public void remove() { it.remove(); }
}
```

Now programs of interest can distinguish between monitored and unmonitored iterators by simply creating `SafeIterators` from `Iterators`. For example, consider the program `Test.java`:

```
public class Test {
    public static void main(String[] args) {
        Vector<Integer> v = new Vector<Integer>();
        v.add(1); v.add(2); v.add(4); v.add(8);
        Iterator it = v.iterator();
        SafeIterator i = new SafeIterator(it);
        int sum = 0;
        if (i.hasNext()) {
            sum += (Integer)i.next();
            sum += (Integer)i.next();
            sum += (Integer)i.next();
            sum += (Integer)i.next();
        }
        System.out.println("sum: " + sum);
    }
}
```

Note that, to build this program, the `javac` and `java` commands require the RV-Monitor runtime library (`rv-monitor-rt.jar`) and the monitor directory to be in the `CLASSPATH`. This allows the use of the RV-Monitor runtime, required by the libraries generated by the `rv-monitor` command. With this in mind, and if the `rvm` directory contains the `HasNext.rvm` property and the corresponding generated library, `HasNextRuntimeMonitor.java`, the commands to compile and run the program above are as follows:

```
$ javac Test.java SafeIterator.java rvm/HasNextRuntimeMonitor.java
$ java Test
```

RV-Monitor, then, outputs the following:

```
! hasNext() has not been called before calling next() for an iterator
   ! hasNext() has not been called before calling next() for an iterator
   ! hasNext() has not been called before calling next() for an iterator
   sum: 15
```

The automated instrumentation method. In some use-cases, the manual insertion of calls to a monitoring library can be tedious and error-prone. In these cases, aspect-oriented programming [17] can be used to instrument large code bases automatically. We can create an AspectJ aspect that calls monitoring methods for all instances of `next` and `hasNext` in the program. This aspect can be weaved throughout any program to make all uses of `Iterators` safe. For example:

```
aspect HasNextAspect {
    after(Iterator i) : call(* Iterator.hasNext())
          && target(i) {
        rvm.HasNextRuntimeMonitor.hasnextEvent(i); }
    after(): before(Iterator i) : call(* Iterator.next())
          && target(i) {
        rvm.HasNextRuntimeMonitor.nextEvent(it); }
}
```

Additionally, the RV-Monitor ecosystem includes a database of over 200 real, production-quality properties specifying the correct operation of the Java and Android APIs that may be automatically checked in Java programs using AspectJ and RV-Monitor.[11] The RV-Monitor distribution provides a pre-compiled suite of common Java API protocol properties together in an agent that is automatically invoked when `java` is replaced with `rv-monitor-all` in the command line.

4.3 Specifying and checking properties with RV-Monitor

In this section, we demonstrate monitors generated by RV-Monitor from properties expressed in three different formalisms (as finite-state machines, regular expressions, and linear temporal logic) to check Java and C/C++ programs.

The finite-state machine (FSM) formalism. First, we explore how we can leverage the FSM formalism to express the Java API property that the `next` method of an iterator must not be called without a previous call to the `hasNext` method. This property can be specified as shown below:

[11] For more information, please see `https://github.com/runtimeverification/property-db`.

```
full-binding HasNext(Iterator i) {
    event hasnext(Iterator i) { } // after
    event next(Iterator i) { }    // before
    fsm :
        start [
            next -> unsafe
            hasnext -> safe ]
        safe [
            next -> start
            hasnext -> safe ]
        unsafe [
            next -> unsafe
            hasnext -> safe ]
    alias match = unsafe
    @match {
        System.out.println(
            "next called without hasNext!");
    }
}
```

After installing RV-Monitor we can see what monitoring of this property looks like in action:

```
$ cd examples/FSM/HasNext
$ rv-monitor rvm/HasNext.rvm
$ javac rvm/HasNextRuntimeMonitor.java HasNext_1/HasNext_1.java
$ java HasNext_1.HasNext_1
```

RV-Monitor reports that `next` has been called without `hasNext` four times in the corresponding Java code.

The FSM formalism: an example in C. In addition to Java, RV-Monitor also supports C. Below is an example of a simple property about the state of a seat belt in a vehicle simulation:

```
SeatBelt {
    event seatBeltRemoved() {
        fprintf(stderr, "Seat belt removed.");
    }
    event seatBeltAttached() {
        fprintf(stderr, "Seat belt attached.");
    }
    fsm :
        unsafe [ seatBeltAttached -> safe ]
        safe [ seatBeltRemoved -> unsafe ]
    @safe {
        fprintf(stderr, "set max speed to user input.");
    }
    @unsafe {
        fprintf(stderr, "set max speed to 10 mph.");
    }
}
```

The extended regular expression (ERE) formalism. Consider the property expressed as a regular expression below. This property aims to ensure there are no writes to a file after the file is closed. As in the first example, this is a property of the Java API that is potentially a source of many program bugs. It can be defined as follows:

```
SafeFileWriter(FileWriter f) {
    static int counter = 0;
    int writes = 0;
    event open(FileWriter f) {    // after
        this.writes = 0;
    }
    event write(FileWriter f) {   // before
        this.writes ++;
    }
    event close(FileWriter f) { } // after
    ere : (open write write* close)*
    @fail  {
        System.out.println("write after close");
        __RESET;
    }
    @match {
        System.out.println(++counter + ":" + writes);
    }
}
```

In the previous examples of property checking, we defined properties of invalid program executions. This example, however, defines a property representing the correct execution of a `FileWriter` in Java. Specifically, this property formalizes the behavior of a file being opened, written to some number times, and then closed. Correct execution traces for a given `File` object contain this sequence of events occurring zero or more times. The above property can be exercised on the code from the RV-Monitor distribution as shown below:

```
$ cd examples/ERE/SafeFileWriter
$ rv-monitor rvm/SafeFileWriter.rvm
$ javac rvm/SafeFileWriterRuntimeMonitor.java \
      SafeFileWriter_1/SafeFileWriter_1.java
$ java SafeFileWriter_1.SafeFileWriter_1
```

The linear temporal logic (LTL) formalism. The same specification about files mentioned above can be captured in another formalism, namely linear temporal logic. The only difference from the last specification is that we replace the ERE property with an LTL property:

```
ltl : [](write => (not close S open))
```

This property specifies that at a `write`, there should not have been in the past a call to `close`, and that `open` must have occurred after the start. This represents the same property as the extended regular expression property and demonstrates the ability of RV-Match to use multiple formalisms, depending on the knowledge and preferences of the property developer.

Analyzing logs. In addition to monitoring software execution, RV-Monitor is able to check logical properties over text-based log files. These properties can be anything that is Turing computable, and do not require storing the entire log files. This makes RV-Monitor ideal for in-depth analysis of large log files which may be impractical to analyze with traditional techniques. For more details, please see the Running Examples section of RV-Monitor documentation [7].

5 Conclusion

Whereas RV-Match is a tool for rigorously detecting all forms of undefinedness, RV-Predict targets the hard problem of efficiently detecting data races. RV-Match interprets programs according to a complete operational semantics, while RV-Predict is able to infer a maximal causal model of concurrent behavior from a single real execution trace. RV-Monitor, on the other hand, confronts the problem of software correctness from a broader perspective by providing a framework for directly monitoring and enforcing adherence to a specification. Together, these tools represent a rigorous yet pragmatic and user-friendly approach to verification (eponymously) characterized by its focus on the analysis of programs at runtime.

References

[1] Grigore Roșu and Traian Florin Șerbănuță. "An Overview of the K Semantic Framework". In: 79.6 (2010), pp. 397–434. doi: 10.1016/j.jlap.2010.03.012.

[2] Traian Florin Șerbănuță, Feng Chen, and Grigore Roșu. "Maximal Causal Models for Sequentially Consistent Systems". In: *RV 2012*. LNCS. doi: 10.1007/978-3-642-35632-2_16.

[3] Dirk Beyer. "Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference (TACAS'16)". In: 2016, pp. 887–904. doi: 10.1007/978-3-662-49674-9_55.

[4] Brian Campbell. "An Executable Semantics for CompCert C". In: *Certified Programs and Proofs*. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 60–75. doi: 10.1007/978-3-642-35308-6_8.

[5] Géraud Canet, Pascal Cuoq, and Benjamin Monate. "A Value Analysis for C Programs". In: *Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE, 2009, pp. 123–124. doi: 10.1109/SCAM.2009.22.

[6] Clang. *Clang 3.9 Documentation*. url: http://clang.llvm.org/docs/index.html.

[7] Philip Daian. *RV-Monitor Documentation*. https://runtimeverification.com/monitor/1.3/docs/. 2015.

[8] Chucky Ellison. "A Formal Semantics of C with Applications". PhD thesis. University of Illinois, July 2012. URL: `http://hdl.handle.net/2142/34297`.

[9] Chucky Ellison and Grigore Roșu. "An Executable Formal Semantics of C with Applications". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 2012, pp. 533–544. DOI: `10.1145/2103656.2103719`.

[10] GrammaTech. *CodeSonar*. URL: `http://grammatech.com/products/codesonar`.

[11] Dwight Guth. *RV-Match Documentation*. `https://runtimeverification.com/match/1.0-SNAPSHOT/docs/`. 2016.

[12] Dwight Guth. *Using RV-Predict to track down race conditions*. `https://runtimeverification.com/blog/?p=47`. 2015.

[13] Chris Hathhorn, Chucky Ellison, and Grigore Roșu. "Defining the Undefinedness of C". In: *36th Conference on Programming Language Design and Implementation (PLDI'15)*. 2015.

[14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Roșu. "Maximal Sound Predictive Race Detection with Control Flow Abstraction". In: *PLDI 2015*. DOI: `10.1145/2594291.2594315`.

[15] ISO/IEC JTC 1, SC 22, WG 14. *ISO/IEC 9899:2011: Prog. Lang.—C*. Tech. rep. International Organization for Standardization, 2012.

[16] Dongyun Jin et al. "JavaMOP: Efficient Parametric Runtime Monitoring Framework". In: *ICSE 2012*. IEEE, June 2012, pp. 1427–1430. DOI: `http://dx.doi.org/10.1109/ICSE.2012.6227231`.

[17] Gregor Kiczales et al. "An overview of AspectJ". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 327–354.

[18] Yilong Li. *Detecting popular data races in Java using RV-Predict*. `https://runtimeverification.com/blog/?p=58`. 2015.

[19] Yilong Li. *RV-Predict Documentation*. `https://runtimeverification.com/predict/1.8.2/docs/`. 2015.

[20] Fred Long et al. *The CERT Oracle secure coding standard for Java*. The SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2012. ISBN: 978-0-321-80395-5.

[21] Qingzhou Luo et al. "RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties". In: *RV 2014*. LNCS, Sept. 2014.

[22] MathWorks. *Polyspace Bug Finder*. URL: `http://www.mathworks.com/products/polyspace-bug-finder`.

[23] MathWorks. *Polyspace Code Prover*. URL: `http://www.mathworks.com/products/polyspace-code-prover`.

[24] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 2007, pp. 89–100. DOI: `10.1145/1250734.1250746`.

[25] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. "Test Suites for Benchmarks of Static Analysis Tools". In: *The 26th IEEE International*

Symposium on Software Reliability Engineering (ISSRE'15). Vol. Industrial Track. 2015.

[26] *Why does this Java program terminate despite that apparently it shouldn't (and didn't)?* `http://stackoverflow.com/questions/16159203/why-does-this-java-program-terminate-despite-that-apparently-it-shouldnt-and-d`. 2013.