








# Java Ranger: Supporting String and Array Operations in Java Ranger (Competition Contribution)\*

\*\*Soha Hussein\*\*\*<sup>1</sup> , Qiuchen Yan<sup>1</sup> , Stephen McCamant<sup>1</sup> ,  
Vaibhav Sharma<sup>1</sup> , and Michael W. Whalen<sup>1</sup> 

University of Minnesota, Minneapolis, MN, USA  
{soha, yanxx297, smccaman, vaibhav, mwwhalen}@umn.edu

**Abstract.** Java Ranger is a path-merging tool for Java Programs. It identifies branching regions of code and summarizes them by generating a disjunctive logical constraint that describes the behavior of the code region. Previously, Java Ranger showed that a reduction of 70% of execution paths is possible when used to merge branching regions of code that support numeric constraints.

In this paper, we describe the support of two additional features since participation in SV-COMP 2020: symbolic array and symbolic string operations. Finally, we present a preliminary evaluation of the effect of the structure of the disjunctive constraint on the solver's performance. Results suggest that certain constraint structures can speed up the performance of Java Ranger.

## 1 Introduction

Path-merging [1,7,8] is a technique that speeds up the execution of Dynamic Symbolic Execution (DSE) by collapsing paths within code regions into a disjunctive logical constraint. Java Ranger (JR) [12] is a path-merging tool for Java Programs. It summarizes symbolic branches during execution. JR generates the disjunctive logical constraint for a code region predicated on a symbolic branch by using a sequence of transformations. For example, JR alternates between substituting values for local variables in its summary and inlining method summaries to eliminate dynamically dispatched method invocations. See [11] for more information.

## 2 Path Merging Extensions and Results

Despite handling many of the Java language features, in SV-COMP 2020 [10] JR did not support symbolically executing string functions. It also did not sum-

\* The research described in this paper has been supported in part by the National Science Foundation under grant 1563920, and Google Summer of Code.

\*\* Jury member

\*\*\* Lecturer on a Leave of Absence Ain Shams University, Cairo, Egypt  
soha.hussien@cis.asu.edu.eg

marize `arrayload` and `arraystore` statements that exist outside a code region predicated on a symbolic branch. For example, if  $a$  and  $i$  are symbolic integers, JR could summarize a region of the form:  $if(a) \{myval = arr[i]...\}$  But not:  $myval = arr[i]$ . More precisely, the newly introduced features to JR include:

1. **Summarizing Array Creation of Symbolic Size:** to support the creation of symbolic-sized single and multi-dimensional arrays, we bound the symbolic size to several values, and we executed the program on each concrete value.
2. **Summarizing ArrayLoad and ArrayStore:** to support the arrayload and the arraystore of a symbolic index, we create a disjunctive constraint that describes possible valuations. This constraint is then pushed on the path condition. For example: for a symbolic index  $i$  and an array  $arr$  of size 3, we encode arrayload of the form  $myval = arr[i]$  as

$$myval := ite(i == 0, arr[0], ite(i == 1, arr[1], arr[2]))$$

Similarly, we encode the arraystore of the form  $arr[i] = myval$  as

$$\begin{aligned} arr[0]_{new} &:= ite(i == 0, myval, arr[0]_{old}) \\ \wedge arr[1]_{new} &:= ite(i == 1, myval, arr[1]_{old}) \\ \wedge arr[2]_{new} &:= ite(i == 2, myval, arr[2]_{old}) \end{aligned}$$

where  $arr[i]_{old}$ , and  $arr[i]_{new}$  indicate the old and the new values of the array  $arr$  at index  $i$ .

3. **Symbolically Executing Symbolic Strings:** We added support to some basic string operations for the `String` package and the `StringBuilder` package; this includes but is not limited to `charAt`, `concat`, `contains`, `endsWith`, `equals`, `indexOf`, `length`, `replace`, `startsWith`, `isEmpty` and `substring`.

## 2.1 Run Configuration

In addition to JR configurations used in SV-COMP 2020 [10], we used the below configurations for turning on the added features.:

- `symbolic.jarrays=true`: to enable the above array features.
- `symbolic.strings=true`: to enable executing symbolic string
- `symbolic.string_dp=z3str3`: to use Z3's default string theory.
- `symbolic.string_dp_timeout_ms=3000`: for timeout on the string queries.

## 2.2 Results

To understand the value of the JR’s extensions above, we evaluated the old JR tool [9] from SV-COMP 2020, which had no support for symbolic arrays nor symbolic strings, to JR’s version participating in 2023. We ran both versions on the verification tasks used in SV-COMP 2023. Results in Tb. 1 show an increased number of correctly solved tasks from 429 to 475, but more importantly, a significant reduction in incorrect results from 97 to zero. These improved scores show the importance and significance of the added support.

Unfortunately, however, because the current version of JR has no support for witness generation, all correctly reached false verdicts were not included in the SV-COMP 2023 score [2], which resulted in JR scoring 400 points instead of 675. In the future, we plan to extend JR to support witness generation.

	JR 2020	JR 2023
<b>number of tasks</b>	587	
<b>total correct</b>	429	475
correct true	220	200
correct false	209	275
<b>total incorrect</b>	97	0
incorrect true	97	0
incorrect false	0	0
<b>Score</b>	<b>-2455</b>	<b>400</b>

Table 1: results of JR’s version participating in 2020 versus the improved 2023 version

## 3 Formula Structure in Path-Merged String Constraints

Fig. 1 shows `loopCharAt`: an SV-COMP 2023 verification task [3] (from an example of Avgerinos et al. [1]) that can dramatically benefit from path-merging. The task accepts a symbolic string `arg`, and checks each character to see if it is the letter ‘B’. If so it increments `counter`. The assertion fails if the value of the counter can be 121. For a symbolic string of length  $n$ , this code has  $2^n$  execution paths, since each character can be B or not B independently. But applying path merging to the `if` statement leads to a single execution path for a given length string. While JR sees this expected asymptotic benefit (one path per string length), reaching the assertion failure takes more than 2 hours, well beyond the competition time limit. Most time is spent in the solver, so we investigated whether changing the syntax of the query could improve performance.

```
public static void loopCharAt(String arg) {
    int counter = 0;
    for (int i = 0; i < arg.length(); i++) {
        char myChar = arg.charAt(i);
        if (myChar == 'B') counter++;
    }
    assert (counter != 121);
}
```

Fig. 1: `loopCharAt` Example

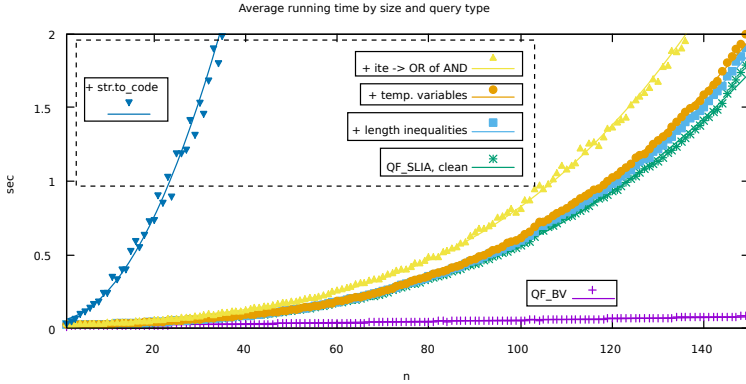


Fig. 2: Average running time by size and query type

Each query generated from the satisfiability of the assert statement asks whether an  $n$ -character string can contain 121 (or more generally,  $k$ ) B characters; this query is satisfiable if  $0 \leq k \leq n$ . We used a script to generate variations of the query for different values of  $n$  and  $k$ , and different semantically equivalent ways of expressing the constraints. We then measured the time to solve the queries using Z3 4.8.15 with the `seq` string solver, on an Intel i7-3770 workstation running Ubuntu 20.04. The choice of  $k$  appeared to have little effect on performance, so we report the results of averaging over runs with  $0 \leq k \leq n + 1$ . Figure 2 shows how the running time grows with  $n$ , and that the query style has a large impact on performance.

We describe the query styles in order of increasing overhead. Because no complex string operations are needed, an equivalent query can be expressed in a simple bit-vector (QF\_BV) logic. This was by far the fastest, and the only style where the running time appears to grow linearly with  $n$ . The remaining styles use a logic of strings and integers (QF\_SLIA), and we started with the constraint style that seemed most natural to write by hand (“clean”) and sequentially added complexities to make the constraints increasingly similar to those JR produces. All these QF\_SLIA styles appear to slow down as a cubic polynomial in  $n$ , as illustrated by the best-fit lines. Two features of JR’s queries had little effect on performance: expressing the string length with a series of inequalities (in JR these come from the loop), and introducing a temporary variable corresponding to each update of the counter. A modest but measurable slowdown came from expressing the effect of the merged region with OR and AND operations, instead of the functional if-then-else operator. A final dramatic slowdown came from constraining the value of each character via its character code (`= (str.to_code (str.at s 0)) 66`) (natural because Java’s `char` is an integer type) instead of as a one-character string (`= (str.at s 0) "B"`). These results suggest that this verification task could become feasible in 15 minutes if either JR or solvers can transform the slow-to-solve forms into fast-to-solve ones.

## 4 Data-Availability Statement

Java Ranger is developed at the University of Minnesota. It is continuously maintained on GitHub [6]. Readers interested in the reproducibility of Java Ranger results in the competition an artifact can be found here [5,4].

## References

1. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: ICSE. pp. 1083–1094. ACM, New York, NY, USA (2014)
2. Beyer, D.: Results of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). <https://doi.org/10.5281/zenodo.7627787>
3. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2023 and Test-Comp 2023). Zenodo (2023). <https://doi.org/10.5281/zenodo.7627783>
4. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). <https://doi.org/10.5281/zenodo.7627829>
5. Hussein, S., Yan, Q., Sharma, V., McCamant, S., Whalen, M., Visser, W.: Java ranger artifact for sv-comp2023 (2023). <https://doi.org/10.5281/zenodo.7467038>
6. Java Ranger, <https://github.com/vaibhavbsharma/java-ranger>, accessed: 2022-12-17
7. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: PLDI. pp. 193–204. ACM, New York, NY, USA (2012)
8. Sen, K., Necula, G., Gong, L., Choi, W.: MultiISE: Multi-path symbolic execution using value summaries. In: ESEC/FSE. pp. 842–853. ACM (2015)
9. Sharma, V., Hussein, S., Whalen, M., McCamant, S., Visser, W.: Artifact for sv-comp2020 verifiers including java ranger’s (2020). <https://doi.org/10.5281/zenodo.3630205>
10. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger at SV-COMP 2020 (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 393–397. Springer International Publishing, Cham (2020)
11. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In: ESEC/FSE. p. 123–134. ACM, New York, NY, USA (2020)
12. Sharma, V., Whalen, M.W., McCamant, S., Visser, W.: Veritesting challenges in symbolic execution of Java. SIGSOFT Softw. Eng. Notes **42**(4), 1–5 (Jan 2018)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

