Mutation-Based Lifted Repair of Software Product Lines (Artifact)

Aleksandar S. Dimovski
Mother Teresa University, Skopje, North Macedonia

Abstract

In this work, we describe the installation, usage, and evaluation results of the tool SPLAllRepair, which is introduced by the paper "Mutation-based Lifted Repair of Software Product Lines". We provide step-by-step instructions on how to download, run, and compare the tool's outputs to outputs described in the paper. The tool implements a novel lifted repair algorithm for program families (Software Product Lines – SPLs) based on code mutations. The inputs of our algorithm are an erroneous SPL and a specification given in the form of assertions. We use variability encoding to transform the given SPL into a single program, called family simulator, which is translated into a set of SMT formulas whose conjunction is satisfiable iff the simulator (i.e. the input SPL) violates an assertion. We use a predefined set of mutations applied to feature and program expressions of the given SPL. The algorithm repeatedly mutates the erroneous family simulator and checks if it becomes (bounded) correct. The outputs are all minimal repairs in the form of minimal number of (feature and program) expression replacements such that the repaired SPL is (bounded) correct with respect to a given set of assertions. We present the experimental results showing that our approach is able to successfully repair various interesting `#ifdef`-based C SPLs.

2012 ACM Subject Classification Software and its engineering → Software product lines; Theory of computation → Abstraction

Keywords and phrases Program repair, Software Product Lines, Code mutations, Variability encoding

Digital Object Identifier 10.4230/DARTS.10.2.5

Related Article Aleksandar S. Dimovski, "Mutation-Based Lifted Repair of Software Product Lines", in 38th European Conference on Object-Oriented Programming (ECOOP 2024), LIPIcs, Vol. 313, pp. 12:1–12:24, 2024.
`https://doi.org/10.4230/LIPIcs.ECOOP.2024.12`

Related Conference 38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2024 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

In this work, we present a tool, called SPLAllRepair, for lifted (SPL) repair [6] of program families (SPLs) in C [3, 4, 5, 8, 9]. Our proof-of-concept implementation is built on top of the AllRepair tool [11] for repairing single programs. The pre-processor `VarEncode` procedure, which tranforms program families to single programs (called family simulators), is implemented in Java, while the translation and mutation procedures are implemented by modifying the CBMC model checker [1] written in C++, where variability-specific mutations are defined. Moreover, we have experimented by defining various mutations to other types of program expressions (see below). The repair phase is implemented by calling the AllRepair tool [11] written in Python. We also call the MiniCard SAT solver [10] and the Z3 SMT solver [2]. The altered CBMC (plus ∼1K LOC) takes as input a family simulator, and generates a `gsmt2` file containing SMT formulas for all possible mutations of the corresponding statements in the input program. The AllRepair (∼2K LOC) takes as input a `gsmt2` file, generates formulas for SAT and SMT solving, and handles all calls to them.

The tool accepts programs written in C with `#ifdef/#if` directives [3, 4, 5, 8, 9]. It uses three main parameters: *mutation level* that defines the kind of mutations that will be applied to feature and program expressions; *unwinding bound* b that shows how many times loops and recursive functions will be inlined; and *repair size* r that specifies how many mutations will be applied at most to buggy programs. We use two mutation levels: *level 1* contains simpler mutations that are often sufficient for repairment, while *level 2* contains all possible mutations we apply. For example, for arithmetic operators in mutation level 1 we have two sets $\{+, -\}$ and $\{*, \%, \div\}$, which means that $+$ is replaced with $-$ and vice versa, and $*, \%, \div$ can be replaced with each other. On the other hand, in mutation level 2 we have one set $\{+, -, *, \%, \div\}$, which means that any arithmetic operator from the set can be replaced with any other.

We compare three approaches for SPL repair:

- SPLALLREPAIR$_1$ that uses mutation level 1 (a predefined set of simpler mutations applied to feature and program expressions).
- SPLALLREPAIR$_2$ that uses mutation level 2 (a predefined set of richer mutations applied to feature and program expressions).
- `Brute-force` approach that applies the single-program repair tool ALLREPAIR to all individual variants of a program family one by one.

The evaluation is performed on 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM Ubuntu 22.04.3 LTS, with 8 GB memory.

2 MD5 sum of the artifact

f2ecaff1a457cd1f62a152186750634e

3 Content

The artifact package includes:

- xubuntu.ova is a Virtual Machine image containing the tool already installed. Username: tool, Password: tool. Enter 'SPLAllRepair' subfolder of the 'home' folder and follow instructions for using the tool.
- SPLAllRepair.tar.gz contains the tool and instructions how to install and use it. To install it by using step-by-step written commands see HowToInstall.txt.

4 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at [7]: `https://zenodo.org/record/12588017`.

5 Tested platforms

All experiments are executed on a 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM Ubuntu 22.04.3 LTS, with 8 GB memory.

6 License

The artifact is available under license "CC-BY"; http://creativecommons.org/licenses/by/3.0/.

Table 1 Performance results of SPLALLREPAIR$_1$ vs. SPLALLREPAIR$_2$ vs. `Brute-force`. All times in sec.

Benchmarks	$\lvert\mathbb{F}\rvert$	`LOC`	SPLALLREPAIR$_1$			SPLALLREPAIR$_2$			`Brute-force`		
			`Fix`	`Space`	`Time`	`Fix`	`Space`	`Time`	`Fix`	`Space`	`Time`
`intro1`	2	20	✓	7	0.252	✓	25	0.304	✓	5	0.981
`feat-inter`	3	20	×	7	0.254	✓	25	0.315	×	9	2.110
`feat_power`	1	20	✓	16	0.722	✓	403	7.79	×	8	0.882
`factorial`	2	50	✓	86	2.540	✓	1603	107.3	✓	81	4.196
`sum`	2	30	✓	17	0.376	✓	266	2.656	✓	18	1.147
`sum_mton`	1	20	✓	32	0.770	✓	681	15.22	×	10	0.556
`4-A-Codeflaws`	2	95	×	52	0.426	✓	1390	2.578	×	36	1.180
`651-A-Codeflaws`	2	85	✓	180	3.394	✓	2829	38.53	✓	237	5.78
`tcas_spl1`	1	305	×	37	0.99	✓	158	6.10	×	37	1.41
`tcas_spl2`	1	305	×	38	1.19	✓	164	8.94	×	38	1.47
`Qlose_multiA`	3	32	×	122	0.711	✓	5415	69.21	×	65	5.781
`Qlose_iterPower`	2	30	×	9	0.973	✓	38	2.921	×	16	1.391
`MinePump_spec1`	6	730	✓	38	300.0	$\underline{✓}$	-	timeout	$\underline{✓}$	-	timeout
`MinePump_spec3`	6	730	✓	39	291.0	$\underline{✓}$	-	timeout	$\underline{✓}$	-	timeout

7 Size of the artifact

9.37 GiB

A Performance results

For step-by-step execution of all test cases of the tools follow: `ExperimentalResultsTable1.txt` and `ExperimentalResultsTable2.txt`.
Enter the folder that contains the tool:
`cd SPLAllRepair/scripts`

A.1 Table 1

We first present results shown in Table 1 on pp. 16 in the paper [6] (here reproduced in Table 1). Table 1 shows performance results of running SPLALLREPAIR$_1$, SPLALLREPAIR$_2$, and the `Brute-force` approach on the given benchmarks. We use mutation level 1 for `Brute-force`. Note that the `Brute-force` approach calls translation, mutation, and repair procedures for each variant separately, whereas SPLALLREPAIR$_1$ and SPLALLREPAIR$_2$ call these procedures only once per program family. For each approach, there are three columns: "`Fix`" that specifies with ✓ (resp., ×) whether the given approach finds (resp., does not find) a correct repair for a given benchmark; "`Space`" that specifies how many mutants have been inspected (explored); and "`Time`" that specifies the total time (in seconds) needed for the given tasks to be performed.

A.1.1 SPLAllRepair1

To run SPLALLREPAIR$_1$, we write the following commands.

```
$ ./spl1.sh
$ ./SPLAllRepair.sh Examples/minepump -m 1 -u 5 -s 1 2>&1 | ./ParseResults.sh
```

Table 2 Performance results of SPLALLREPAIR$_1$ for different values of the unwinding bound $u = 2, 5, 8$. All times in sec.

Benchmarks	$u = 2$		$u = 5$		$u = 8$	
	`Fix`	`Time`	`Fix`	`Time`	`Fix`	`Time`
`feat_power`	×	0.254	✓	0.722	✓	0.978
`factorial`	×	1.231	✓	3.540	✓	6.524
`sum`	×	0.304	✓	0.376	✓	0.456
`sum_mton`	×	0.589	✓	0.770	✓	0.922
`651-A-Codeflaws`	✓	1.814	✓	3.394	✓	6.828

Output of the script for each benchmark are two files: a csv file with results summarized in a table, and a text file with all found repairs and the elapsed time until each of them was found. The files can be found in the "RepairResults" folder created under the "scripts" folder. The filename of the results csv and text files will be Repair_results_<settings>_<current_date_and_time>.csv and Repair_results_<settings>_<current_date_and_time>.

▶ Remark. If we want results for all benchmarks from Table1 to be given in one .csv and .txt file, then the files from "minepump" folder: minepump_spec1.c and minepump_spec3.c should be copied to the folder "Table1" and we run the script: `$ ./spl1.sh`

A.1.2 SPLAllRepair2

To run SPLALLREPAIR$_2$, we write the following commands.

```
$ ./spl2.sh
$ ./SPLAllRepair.sh Examples/minepump -m 2 -u 5 -s 1 -t 360 2>&1 |
./ParseResults.sh
```

The files can be found in the "RepairResults" folder created under the "scripts" folder.

▶ Remark. If we want results for all benchmarks from Table1 to be given in one .csv and .txt file, then the files from "minepump" folder: minepump_spec1.c and minepump_spec3.c should be copied to the folder "Table1" and we run the script: `$ ./spl2.sh`

A.1.3 Brute force approach

To run the `Brute force` approach on the examples from the paper [6] (see Table 1), we write the following commands.

```
$ ./spl3.sh
$ ./SPLAllRepair.sh Examples/minepumpall --brute-force -m 1 -u 5 -s 1 -t 360
2>&1 | ./ParseResults.sh
```

The files can be found in the "RepairResults" folder created under the "scripts" folder.

▶ Remark. If we want results for all benchmarks from Table1 to be given in one .csv and .txt file, then the subfolders from the "minepump" folder: minepump_spec1 and minepump_spec3 should be copied to the folder "Table1all" and we run the script: `$ ./spl3.sh`

A.2 Table 2

We now present results shown in Table 2 on pp. 17 in the paper [6] (here reproduced in Table 2). Table 2 shows performance results of running SPLALLREPAIR$_1$ on a selected set of benchmarks for different unwinding bounds u. Recall that our approach reasons about loops by unrolling (unwinding) them, so it is sensitive to the chosen unwinding bound. By choosing larger bounds

u, we will obtain more precise results (more genuine repairs), but we will also obtain longer SMT formulas and slower speeds of the repairing tasks. We can see that the running times of all repairing tasks grow with the number of bound u. Of course, we will also obtain more precise results for bigger values of u, and less precise results (i.e., some genuine repairs will not be reported) for smaller values of u. Hence, there is a preision/speed tradeoff when choosing the unwinding bound b. We obtain similar results for SPLAllRepair$_2$ and the `Brute-force`.

We run SPLAllRepair$_1$ and compare three unwinding bounds: $u = 2$, $u = 5$ (default used in Table 1), $u = 8$.

A.2.1 u=2

```
$ ./SPLAllRepair.sh Examples/Table2 -m 1 -u 2 -s 2 2>&1 | ./ParseResults.sh
```
See the results in “RepairResults” folder.

A.2.2 u=5

```
$ ./SPLAllRepair.sh Examples/Table2 -m 1 -u 5 -s 2 2>&1 | ./ParseResults.sh
```
See the results in “RepairResults” folder.

A.2.3 u=8

```
$ ./SPLAllRepair.sh Examples/Table2 -m 1 -u 8 -s 2 2>&1 | ./ParseResults.sh
```
See the results in “RepairResults” folder.

References

1 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

2 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

3 Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.

4 Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019*, pages 102–114. ACM, 2019. doi:10.1145/3357765.3359518.

5 Aleksandar S. Dimovski. Ctl* family-based model checking using variability abstractions and modal transition systems. *Int. J. Softw. Tools Technol. Transf.*, 22(1):35–55, 2020. doi:10.1007/s10009-019-00528-0.

6 Aleksandar S. Dimovski. Mutation-based lifted repair of software product lines. In *38th European Conference on Object-Oriented Programming, ECOOP 2024*, volume 313 of *LIPIcs*, pages 36:1–36:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ECOOP.2024.36.

7 Aleksandar S. Dimovski. Tool artifact for “mutation-based lifted repair of software product lines”. *Zenodo*, 2024. doi:10.5281/zenodo.12588017.

8 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Several lifted abstract domains for static analysis of numerical program families. *Sci. Comput. Program.*, 213:102725, 2022. doi:10.1016/J.SCICO.2021.102725.

9 Aleksandar S. Dimovski and Andrzej Wasowski. From transition systems to variability models and from lifted model checking back to UPPAAL. In *Models, Algorithms, Logics and Tools*, volume 10460 of *LNCS*, pages 249–268. Springer, 2017. doi:10.1007/978-3-319-63121-9_13.

10 Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free – (poster presentation). In *Theory and Applications of Satisfiability Testing – SAT 2012 – 15th Int. Conf., Proceedings*, volume 7317 of *LNCS*, pages 485–486. Springer, 2012. doi:10.1007/978-3-642-31612-8_47.

11 Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *FM 2016: Formal Methods – 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 593–611, 2016. doi:10.1007/978-3-319-48989-6_36.