UDC 004.9 **doi: 10.26906/SUNZ.2024.2.074**

Mykhailo Hulevych

National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine

CIDER : ASSISTED AUTOMATION TOOL FOR C++ LIBRARIES TESTING

Abstract. Software testing is one of the most important parts of a product development lifecycle (PDL). Companies face significant risks associated with program failures, including financial and reputational ones, having an interest in investing time and money in new testing and quality assurance practices. The programs written in C++ are known as high performant, but complex due to the language specifics. At the same time automated tools that are available mostly for GUI programs can't help with native modules regression testing. Thus, even partial automation of such processes can have a positive effect on PDL in terms of regression testing effectiveness. The article considers a testing technique of recording and playback of program execution on the component level. The proposed tool generates testing scenarios during the execution of a program, giving the ability to perform manual testing and extend the automated component tests execution database. The experiments showed that such scenarios' code coverage results are at a high level and tend to improve when applying optimization algorithms.

Keywords: test automation, script generation, regression testing, C++, optimization, harmony search, code coverage.

Introduction

The modern world has high security challenges making companies invest in security measures while developing complex systems. The software written in C++ consists of a high number of components interacting with each other via a public interface. In general, such components may exchange data not only in the same process but on different machines, which adds additional complexity to testing. Isolation of a component removes redundant complexity and aims to verify that each part of the whole behaves as expected. In addition, the component tests are usually written in scripting languages other than the original component under test by automation test engineers to aim for the problematic branches of execution and specific input data.

At the same time, automatic tests generation for C++ programs question has been a topic of interest in recent years. While automated unit testing tools and techniques have been widely studied and applied in software engineering, including for languages like Java and Python, the unique characteristics of C++ present additional challenges. While significant progress has been made in automatic unit test generation for C++ programs, challenges remain, including the handling of complex language features, such as templates, macros, and pointer arithmetic, as well as the need to balance test effectiveness, efficiency, and scalability, which one of the key parts of C++ language philosophy.

Recent publications. CIRTUS [1] automated unit testing tool for C++ programs generates test suites for unit testing based on source code analysis and method sequences generation, improves the code coverage by applying the *libfuzzer* to change the values of generated method's arguments. The tool has shown an ability to produce the test suites that achieve a high value of branch and statement coverage (up to 95% on one of the experimental setups).

KLOVER [2] uses symbolic execution technique for test inputs generation achieving high structural coverage of the program under test. The C++ program is compiled into the LLVM [3] bytecode and interpreted by KLOVER tool collecting statistics and sanity information. The experiments were made on 4 industrial programs up to 3k LOC and showed positive results regarding the manual testing. Another tool named PAULS [4] is oriented on Java test case generation. It uses dynamic analysis of program execution to infer a call sequence model with static analysis of call relations based on argument dependence. The study is aimed at existing unit tests optimization in terms of code coverage, test case source code size and redundant cases search. It showed significant code coverage improvement on 6 open-source libraries, improving the existing unit test suites.

At the same time, alternatives suffer from invalid method call sequences and irrelevant input data generation, producing a high tests fault rate.

CIDER

The paper introduces a new approach of automatic testing scenarios generation named CIDER. Instead of earlier mentioned tools it's aimed at a script-based testing approach of C++ programs code base. The generated scripts can be used in component test suites for industrial projects. CIDER uses a test recording approach, meaning that output scenarios are first executed right from a target application that uses the library under test. The generated method calls sequences are relevant to the program execution and possibility of faulty call sequences without meaning is omitted. Then scripts are optimized to achieve higher code coverage and saved in the test suite.

In the next sections, we will overview a general script generation process, as well as valuable implementation details of CIDER tool. The experimental setup includes two C++ open-source libraries: *Hjson* [5] and *Pugixml* [6]. The following research questions will be answered:

RQ1: Are CIDER-recorded scripts executions comparable in terms of code coverage with the original test suites executions?

RQ2: Can generated scenarios be optimized in terms of code coverage?

Process overview

The formal CIDER usage process is shown in Fig. 1 and consists of:

1. Parsing target library *T* header files into an abstract syntax tree (AST) for detecting all functions and parameters with corresponding types.

2. Generation of C++ source wrapper files containing special hooks for function calls data collection.

3. Compile and build the T' library.

4. Using the AST generate the (*.swig) configuration files for SWIG [7] generator for script bindings generation.

5. Call SWIG generator to get SWIG C++ source files with lua scripting language bindings.

6. Compile and build the L library from sources generated on stage 6, which is used by lua interpreter I for the generated scripts execution.

7. Inject T' library into the target application and perform executable running in order to get session output data for script S generation.

8. Generate lua script S.

9. Execute S on lua interpreter I with imported library T and process output library coverage information.

10. Optimize script S using the harmony search metaheuristic algorithm.

11. Save the optimized version S' into the regression tests suite.

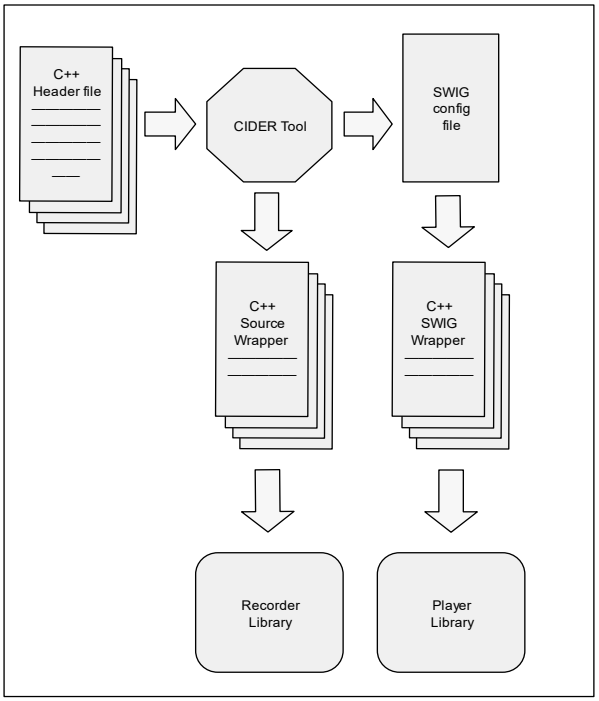

Fig. 1. Formal CIDER process overview

Data model

Every program execution is represented as a test scenario in our model. The test scenario consists of a sequence of method calls with corresponding arguments.

Each argument has one of the supported C++ types and value, recorded during an execution. The supported argument types by CIDER generally are:

- Strings (std::string, const char*, etc.).
- Integers (int, short, unsigned int, etc.).
- Booleans.
- Doubles (float, double).
- User Data (enums, classes, structs, etc.).

After scenario data is recorded it can be either mutated or generated into lua script [8]. Example of the recorded data chunks are shown in Fig. 2.

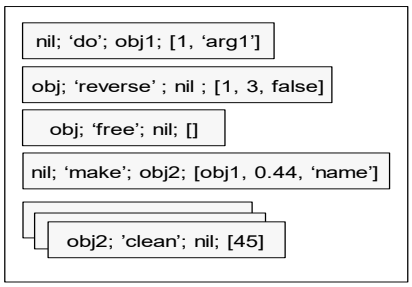

Fig. 2. Recorded program execution data example

Script optimization

An idea from a classic harmony search optimization algorithm (*HS*) was taken, which was first introduced by Z. Geem in 2001 [9]. The reason why it was chosen is that harmony search as a metaheuristic algorithm was compared to others and shown that it's more successful. Also, it's less sensitive to chosen parameters [10]. The task is formulated as finding the maximum value of the coverage function of the given script. The scenario S is represented as sequence of function calls, according to a formula:

$$S = \{F_1, F_2, \dots, F_n\}, \; n \in \mathbb{N}, \tag{1}$$

where *n* is the number of calls.

The F_j can be either a class method call or a free function call having m_j arguments count:

$$F_j = F\left(x_1, x_2, \dots, x_{m_j}\right), \; m_j \in \mathbb{N}. \tag{2}$$

Then, the script S_i can be represented as a function that takes all the arguments from all its constituent functions.

$$S = S(x_1, x_2, \dots, x_l), l = \sum_{j=1}^{n} m_j. \tag{3}$$

The objective function for *HS* algorithm can be represented as:

$$C(S) = C(x_1, x_2, \dots, x_l). \tag{4}$$

The number of the generated scenarios that can be kept in memory equal to k – *harmony memory size.* The *harmony memory* H_m consists of all generated scenarios' arguments and looks like this:

$$H_m = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1l} \\ x_{21} & x_{22} & \cdots & x_{2l} \\ \vdots & \vdots & \ddots & \cdots \\ x_{k1} & x_{k2} & \cdots & x_{kl} \end{pmatrix}, \tag{5}$$

where x_{1i} – arguments of script S_1, $i \in [1 : l]$.

The harmony from the memory is used with θ probability - *memory consideration rate*. During a new harmony generation, each argument can be mutated with ζ *mutation rate*. The mutation formulas for each type of parameters are shown in Table 1.

Table 1 – **Argument mutation formulas**

Argument type	Mutation formula
Integers, Floating points	$I_i' = \mathrm{XOR}(I_i[\mathrm{k}], U_{int}^{0,255}),$ $k = U_{int}^{0,sizeof(T)-1}$
Booleans	$B_i' = U_{int}^{0,1}$
Enums	$E_i = \{e_1, e_2, \dots, e_m\},$ $E_i' = E_i[l], \; l = U_{int}^{0,m}$

The $I_i[k]$ represents a *k*-indexed byte from integer value, where *k* is taken from uniform int distribution in the range from *0* to size of *T* in bytes minus 1. For example, for the *char* type the indexed byte is always equal to 0. The formula was firstly introduced by M. Zalewski and is used in the libFuzzer implementation [11]. For the *boolean* values the mutated value is taken as uniform integer distribution in range from 0 to 1. For the *enum* values, all valid values are gathered into an array E_i, and then indexed via *l*, that is taken as uniform int distribution from 0 to the size of array E_i. For *integer* values. the mutation of parameters doesn't change the number of bytes, which represent the type of the argument. Also, *string* values aren't mutated as it can break the context and make it irrelevant. For example, string parameters can represent JSON-encoded data.

Experimental results

The experimental setup includes two C++ open-source libraries widely used in production codes all over the world. The subjects are represented in Table 2.

Table 2 – **Experimental subjects**

Library	Hash	Sources (LoC)	Test Suites (LoC)
Hjson	f18f03f	3936	1970
Pugixml	b2b4664	8943	8800

The library interfaces were parsed into the AST and source wrappers were generated and compiled into the original test suites. The test suites were run, and test scripts were generated. Example of the script is shown in Fig. 3.

```
1. object_1 = hjson.Value(hjson.UChar(248))
2. object_2 = hjson.Value(hjson.Char(100))
3. object_4 = hjson.plusOp(object_1, object_2)
4. hjson.equalEqualOp(object_4, hjson.Int(350))
5. object_4 = hjson.plusOp(object_2, object_1)
6. hjson.equalEqualOp(hjson.Int(334), object_4)
7. object_4 = hjson.plusOp(hjson.UChar(250), object_1)
8. hjson.equalEqualOp(object_4, hjson.Int(500))
```

Fig. 3. Generated script part example for Hjson library

Code coverage measurement of the original test suite TS_o and CIDER generated script S_c one is shown in Table 3. The values for generated script coverage and the original test suite differ. The reasons were investigated and was found out that it's due to fact that not all functions of C++ nature making sense in the context of lua. For example, move assignment operators and copy assignment operators are ambiguous from the lua side, as r-value references aren't supported by lua.

Table 3 - **Comparison of code coverage results**

Library	Line Cov (%)		Branch Cov (%)		Function Cov (%)	
	TS_o	S_c	TS_o	S_c	TS_o	S_c
Hjson	86,6	82,8	52,3	48,3	50,2	49,3
Pugixml	72,2	68,9	61,1	58,9	68,1	66,3

In addition, the execution time of TS_o and S_c were compared, results are shown in Table 4. The experiment was conducted using a MacBook Pro 2019 equipped with an Intel Core i7 processor (2.6 GHz, 6 cores), 16 GB of DDR4 RAM (2666 MHz), and a 512 GB SSD. The generated scripts were executed step-by-step, and the code coverage was measured. Results are shown in Fig. 4 and 5.

Table 4 – **Comparison of average execution times**

Library	S_c instructions count	Avg. time (ms)	
		TS_o	S_c
Hjson	4178	635	649
Pugixml	17050	57	406

For both libraries there are continuous regions of instructions where the code coverage doesn't grow. Such regions contain similar script instructions with parameters that can be also optimized with *HS* algorithm to achieve the highest code coverage without generating new function calls.

Harmony search optimizations were performed on average with 5k iterations, and most effective parameters are H_m=10; $\theta = 0.5$; $\zeta = 0.2$. The stopping criteria was not finding any new solutions that can be saved into the memory over 300 iterations. Optimized and original script comparison are shown in Table 5, and in Fig. 6 and 7.

Table 5 – **Comparison of code coverage results for optimized and initial scripts**

Library	Line Cov (%)		Branch Cov (%)		Function Cov (%)	
	S_c	S_c^o	S_c	S_c^o	S_c	S_c^o
Hjson	82,8	83,4	48,3	49,3	49,3	50,8
Pugixml	68,9	73,2	58,9	62,8	66,3	67,1

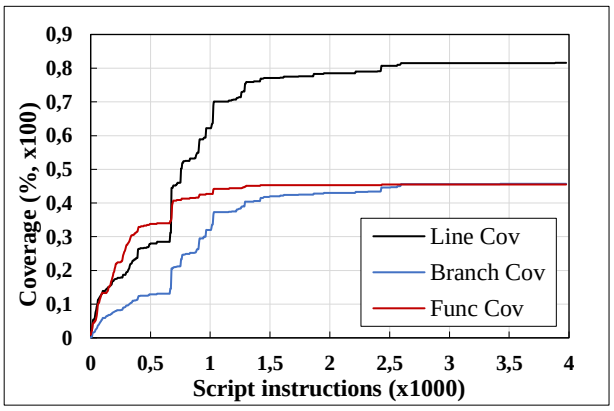

Fig. 4. Hjson coverage script instructions dependence

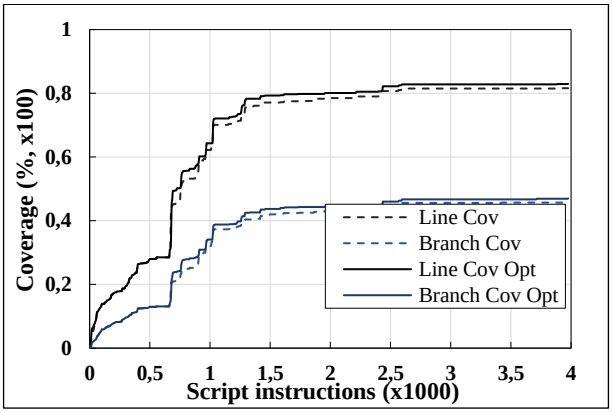

Fig. 6. Hjson optimized script vs original comparison

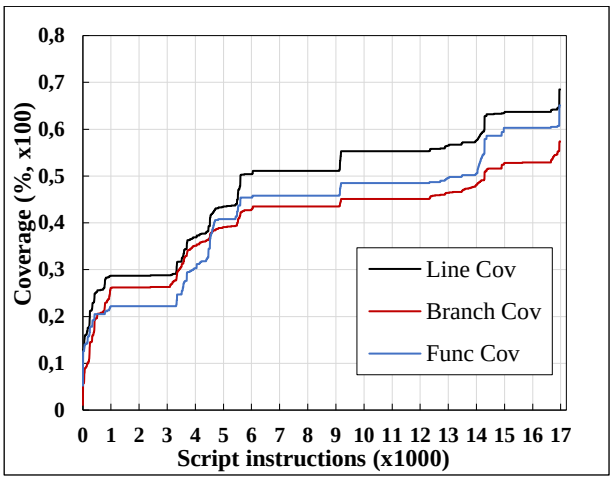

Fig. 5. Pugixml coverage script instructions dependence

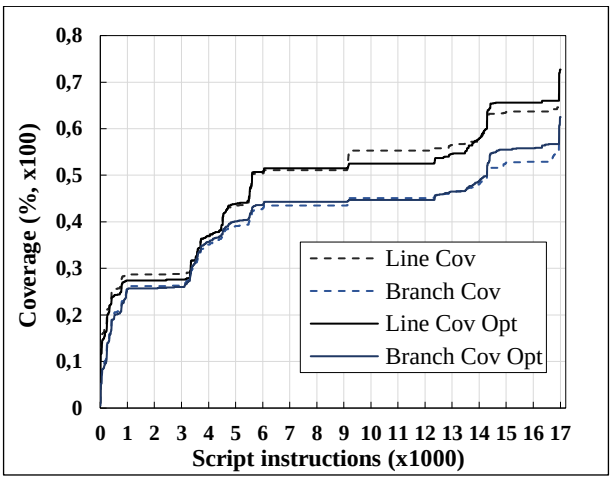

Fig. 7. Pugixml optimized script vs original comparison

Conclusion

A new tool for assisted test scenario generation has been presented. Experiments on third-party libraries have shown that the scenarios and original test suites have comparable code coverage and execution time. In addition, the generated scripts have been optimized using the harmony search algorithm. The result scenarios can be used to improve existing regression testing test suites. Step-by-step coverage measurement has shown the regions of the original test suite that can be improved applying mutation of the parameters. For more covered Hjson the coverage improvement value is less than for the Pugixml library. The technique can be studied and improved further with legal method sequences generation, extending the search space for tests scenarios optimization.

REFERENCES

1. Yunho, K., Kim, M. and Kim, Y. (2022), "CITRUS: Automated Unit Testing Tool for Real-world C++ Programs", *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Valencia, Spain, pp. 400–410, doi: https://doi.org/10.1109/ICST53961.2022.00046
2. Li, G., Ghosh, I. and Rajan, S. P. (2011), "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs", *Computer Aided Verification 23rd International Conference*, Springer, Berlin, Heidelberg, doi: https://doi.org/10.1007/978-3-642-22110-1_49
3. (2024), *The LLVM compiler infrastructure*, available at: http://www.llvm.org/
4. Zhang, S., Saff, D., Bu, Y. and Ernst, M.D. (2011), "Combined Static and Dynamic Automated Test Generation", *ISSTA '11: Proc. of the 2011 Int. Symp. on Software Testing and Analysis*, N-Y, pp. 353–363, doi: https://doi.org/10.1145/2001420.2001463
5. (2024), *Hjson, a user interface for JSON*, available at: https://hjson.github.io
6. (2024), Light-weight, simple and fast XML parser for C++ with XPath support, available at: https://pugixml.org/
7. (2024), *Simplified Wrapper and Interface Generator*, available at: https://www.swig.org
8. (2024), *The Programming Language Lua*, available at: https://www.lua.org
9. Geem, Z., Kim, J. and Loganathan, G. (2001), "A new heuristic optimization algorithm: Harmony search", *Simulation*, vol. 76, no. 2, pp. 60–68, doi: https://doi.org/10.1177/0037549701076002
10. Yang, X.-S. (2009), "Harmony Search as a Metaheuristic Algorithm", *Music-Inspired Harmony Search Algorithm: Theory and Applications, Studies in Computational Intelligence*, vol. 191, Springer Berlin, Editor Z. W. Geem, 2009, pp. 1–14, doi: https://doi.org/10.1007/978-3-642-00185-7_1
11. Zalewski, M. (2024), *American fuzzy lop (2.52b)*, available at: https://lcamtuf.coredump.cx/afl

Received (Надійшла) 27.02.2024
Accepted for publication (Прийнята до друку) 24.04.2024

CIDER : Інструмент автоматизації тестування C++ бібліотек

М. В. Гулевич

Анотація. Програмне тестування є однією з найважливіших частин життєвого циклу розробки продукту. Компанії стикаються з значними ризиками, пов'язаними з відмовами програм, включаючи фінансові та репутаційні, тому цікавляться вкладенням часу та грошей у нові методи тестування та забезпечення якості. Програми, написані на C++, відомі своєю високою продуктивністю, але складні у зв'язку зі специфікою мови. У той же час автоматизовані інструменти, які доступні переважно для програм з графічним інтерфейсом користувача не є ефективними для тестуванням регресії вбудованих модулів. Таким чином, навіть часткова автоматизація таких процесів може мати позитивний вплив у відношенні оптимізації часу тестування регресії. У статті розглянута техніка тестування запису та відтворення виконання програми на рівні компонентів. Запропонований інструмент генерує сценарії тестування під час виконання програми, надаючи змогу виконувати ручне тестування та одночасно розширювати базу даних автоматизованого виконання компонентних тестів. Експерименти показали, що результати покриття коду таких сценаріїв є на високому рівні і мають тенденцію до покращення через алгоритмічну оптимізацію.

Ключові слова: автоматизація тестування, генерація скриптів, регресивне тестування, C++, оптимізація, гармонічний пошук, покриття коду.