CCMOP: A Runtime Verification Tool for C/C++ Programs*

Yongchao Xing[1,2], Zhenbang Chen[1,2](✉)[0000−0002−4066−7892], Shibo Xu[1,2], and Yufeng Zhang[3][0000−0001−6082−4501]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, Changsha, China
[3] College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
{xingyc0979, zbchen}@nudt.edu.cn, yufengzhang@hnu.edu.cn

Abstract. Runtime verification (RV) is an effective lightweight formal method for improving software's reliability at runtime. There exist no RV tools specially designed for C++ programs. This paper introduces the first one, *i.e.*, CCMOP, which implements an AOP-based RV approach and supports the RV of general properties for C/C++ program. CCMOP provides an AOP language specially designed for C++ program to define the events in RV. The instrumentation of RV monitor is done at AST-level, which improves the efficiency of compilation and the accuracy of RV. CCMOP is implemented based on JavaMOP and an industrial-strength compiler. The results of extensive experiments on 100 real-world C/C++ programs (5584.3K LOCs in total) indicate that CCMOP is robust and supports the RV of real-world C/C++ programs.

Keywords: Runtime Verification · C/C++ · Instrumentation · AOP.

1 Introduction

Runtime verification (RV) [19] is a lightweight formal method for verifying program executions. Different from the traditional formal verification methods, such as model checking [13] and theorem proving [15,21], which verify the whole behavior of the program and often face state explosion problem [14] or need labor-intensive manual efforts, runtime verification only verifies a trace (execution) of the program. When the program $\mathcal{P}$ is running, runtime verification techniques collect $\mathcal{P}$'s running information and usually abstract the information into events. A program *trace* t is an *event sequence*. Then, the verification is carried out on the fly for the trace with respect to a formal property φ, *e.g.*, a line-time temporal logic (LTL) property; If t does not satisfy φ [26], the runtime verification

* CCMOP is available at https://rv-ccmop.github.io. Zhenbang Chen is the corresponding author.

will take some efforts, *e.g.*, reporting a warning or error and terminating $\mathcal{P}$ in advance. In this way, runtime verification does not suffer from the scalability problem of traditional formal verification techniques.

Until now, there already exist many runtime verification tools [6] for different program languages, *e.g.*, RTC [23] and E-ACSL [30] for C programs, JavaMOP [8] and TraceMatches [4] for Java programs, to name a few. Usually, a runtime verification tool accepts a program $\mathcal{P}$ and a property φ. Then, the tool automatically generates a runtime monitor $\mathcal{M}$ for φ and instruments the monitor into $\mathcal{P}$. When the instrumented version of $\mathcal{P}$ is executed, $\mathcal{M}$ will online verify $\mathcal{P}$'s trace with respect to φ. Existing runtime verification tools differ in different aspects, including the target program's language, implementation mechanisms (*e.g.*, instrumentation and VM-based approaches), supported verification properties (*e.g.*, LTL, FSM, and CFG), *etc.* These tools are widely applied in different areas and backgrounds [19], which shows the effectiveness of runtime verification.

However, the RV tools for C++ programs are still in demand. Although there exist some sanitizer tools [17] of the LLVM platform [20] that instrument the monitor at the intermediate representation (IR) level, they can only monitor memory-specific properties. Besides, binary-level instrumentation-based RV tools, in principle, support C++ programs, but they also suffer the problems of overhead, across-platform, and inaccuracy [27,32]. As far as we know, there does not exist a runtime verification tool specially designed for C++ programs that supports general properties and instrument monitors at the source code level.

This paper presents CCMOP, *i.e.*, a runtime verification tool for C/C++ programs following the design of JavaMOP [8]. The runtime monitor of the property is first generated in an aspect-oriented programming (AOP) language. We have designed and implemented an AOP platform for C/C++ programs to support the automatic instrumentation of runtime monitors. Monitors can be transparently woven into the program during program compilation. The weaving is carried out at the source code level and on the program's abstract syntax tree (AST). We have implemented the AOP platform based on Clang [12], *i.e.*, an industrial-strength compiler. We have applied CCMOP for 100 real-world C/C++ programs to evaluate our tool. The results indicate that CCMOP can support the runtime verification of real-world C/C++ programs for general properties. The main contributions are as follows.

- We have implemented a runtime verification tool for C/C++ programs that supports the RV of general properties. As far as we know, CCMOP is the first source-level instrumentation-based RV tool for C++ programs.
- We have applied CCMOP for 100 real-world C/C++ programs (5584.3K LOCs in total) with standard C++ language features. The experimental results indicate that CCMOP supports the RV for large-scale C/C++ programs. The runtime overhead of CCMOP on C++ programs is 88% for the use-after-free property.

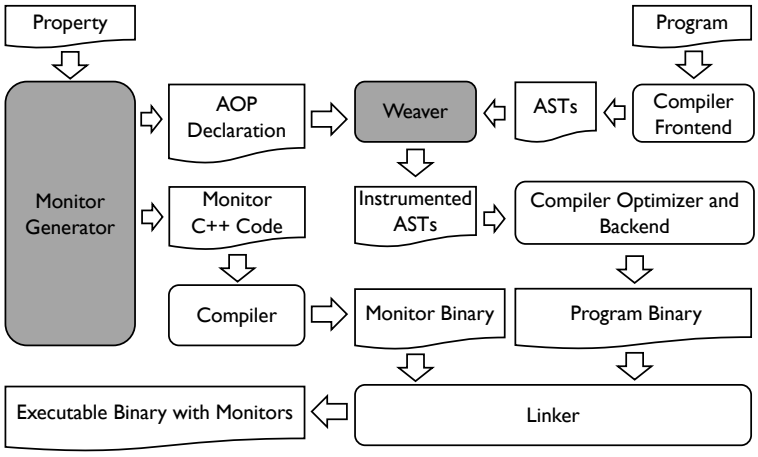

Fig. 1. CCMOP's framework.

2 Framework

Figure 1 shows the framework and basic workflow of CCMOP. The inputs are a property φ and a program $\mathcal{P}$, and the output is the executable binary of the program in which monitors are instrumented. The workflow can be divided into two stages: monitor generation and monitor weaving.

Monitor Generation CCMOP adopts the RV framework of JavaMOP [8] that utilizes AOP for property specification and monitor instrumentation. The property syntax is a variant of JavaMOP's MOP syntax for C/C++ programs. A property φ is composed of the following two parts.

- The declarations of the events, defined by different AOP pointcuts [18], and the AOP language is explained in Section 3. Besides, we can also specify the C/C++ code that will be executed when the event is generated. For example, the following defines event `create` for each `new` statement in the C++ program, and the event `create` is generated after the execution of the `new` statement.

```
event create after(void* key):expr(new *(...))&&result(key) {}   (1)
```

- The formal property defined on the event level, which can be specified by different formalisms, including a logic (*e.g.*, LTL), an automaton (*e.g.*, FSM), and a regular expression, *etc.* The property gives the specification that the program should satisfy during program execution. If the execution satisfies or does not satisfy the property, some operations can be carried out. These operations can be C/C++ statements that are also given in the property.

Based on the property φ, CCMOP automatically generates two parts. The first part is the **AOP declarations** that are in charge of generating events, which are generated with respect to φ's event definitions. The second part is the C/C++ code of the runtime monitor, which is generated with respect to φ's property. The key idea of generating runtime monitors is to generate an automaton for runtime verification according to φ's formal property [7]. Besides, monitor code also contains the event interface methods that interact with system execution.

Monitor Weaving. The second stage happens at the compilation of $\mathcal{P}$. The main job is to instrument the event generation code at the appropriate places of $\mathcal{P}$, accomplished by the **Weaver**. According to the AOP declarations $\mathcal{D}$ generated at the first stage, **Weaver** finds the matched $\mathcal{P}$'s statements of the pointcuts in $\mathcal{D}$ on $\mathcal{P}$'s AST. Then, the event generation code is inserted before or after the statements according to the requirements of the event declarations in φ. Here, the instrumentation is carried out directly on AST. Event generation code invokes the event interface method in the monitor code to notify the monitor that an event is generated and the runtime checking needs to be carried out.

For example, the following code shows the source code after weaving the code for the event `create`. There exists a statement that the pointer `p` is assigned with the address returned by a `new` statement. After instrumentation, the pointer `p` will be passed as the parameter of the event `create`'s interface method invocation.

```
int main(){
  int *p = new int;
  int *key_1=p;
  __RVC_UAF_create(key_1);
  return 0;
}
```

After weaving, the instrumented AST will be used for compiler optimization and code generation in the later compilation stages. Finally, a binary that can generate events for RV is generated. After getting the binary that can generate events, we need to link the binary with the monitor binary that does the real verification job. The executable binary with runtime monitors (denoted by $\mathcal{P}_m$) is finally generated. Then, we can run $\mathcal{P}_m$ with different inputs, and the runtime monitors will carry out the operations defined in φ when φ is violated or satisfied.

In principle, our framework provides an online synchronous approach to runtime verification. The instrumentation for RV is carried out on AST, which enjoys the advantages including lower-overhead, across-platform, accuracy, *etc.*

3 Design and Implementation

AOP Language for C/C++. Figure 2 shows the critical parts of the AOP language for C/C++. The language is based on AspectJ [18], and the figure only shows the abbreviated version for the sake of brevity, where ϵ represents the empty string, $\langle \mathsf{ID} \rangle$ represents an identity name, $\langle \mathsf{IDs} \rangle$ represents a comma dotted $\langle \mathsf{ID} \rangle$ sequence. The particular syntax elements for C/C++ are as follows.

```
⟨Advice⟩ ::= advice ⟨SPointCut⟩⟨VPointCuts⟩:(before | after | around)(⟨VarDecl⟩)
⟨SPointCut⟩ ::= call(⟨FuncDecl⟩) | expr(⟨CExprDecl⟩) | deref (⟨ScopedTypes⟩) |
                end() | ⟨SPointCut⟩ || ⟨SPointCut⟩
⟨VPointCuts⟩ ::= ϵ | && ⟨VPointCut⟩ | && ⟨VPointCut⟩ ⟨VPointCuts⟩
⟨VPointCut⟩ ::= args(⟨IDs⟩) | result(⟨ID⟩) | target(⟨ID⟩) | within(⟨SpaceDecl⟩)
⟨VarDecl⟩ ::= ϵ | ⟨ScopedType⟩ ⟨ID⟩ | ⟨ScopedType⟩ ⟨ID⟩,⟨VarDecl⟩
⟨FuncDecl⟩ ::= (% | ⟨ScopedType⟩) (⟨ScopedType⟩.⟨ID⟩ | ⟨ID⟩)(⟨ScopedTypes⟩ | ...)
⟨CExprDecl⟩ ::= new ⟨ExprDecl⟩ | delete ⟨ExprDecl⟩ | ⟨ExprDecl⟩
⟨ExprDecl⟩ ::= (⟨ScopedType⟩::* | ⟨ScopedType⟩ | *)(⟨ScopedTypes⟩ | ...)
⟨SpaceDecl⟩ ::= ⟨ScopedType⟩ | ⟨ScopedType⟩()
⟨ScopedTypes⟩ ::= ⟨ScopedType⟩ | ⟨ScopedType⟩,⟨ScopedTypes⟩
⟨ScopedType⟩ ::= ⟨BasicType⟩ | ⟨BasicType⟩::⟨ScopedType⟩ | ⟨ScopedType⟩*
⟨BasicType⟩ ::= ⟨ID⟩ | ⟨ID⟩<⟨ScopedTypes⟩> | ⟨PrimitiveTypes⟩
```

Fig. 2. The core syntax of the AOP language.

- We introduce `deref` to match the pointer dereferences in C/C++ programs. Here is an example of the pointcut for matching the dereferences of all string pointers: `deref(std::basic_string<char> *)`.
- To support the namespace and template mechanisms in C++, we introduce ⟨ScopedType⟩, which is also compatible with matching the functions and types of C programs. Besides, we also introduce `cexpr` for matching the object management statements in C++ programs, including class constructors, `new` and `delete` statements. Furthermore, `call(`⟨FuncDecl⟩`)` also supports the matching of the operator functions in C++ program (*e.g.*, `operator+(...)`), and the details are omitted for the sake of brevity.

Similar to JavaMOP, ⟨Advice⟩ is used to capture the monitored objects. For example, the AOP declaration for the event `create` in (1) is as follows.

```
advice expr(new *(...)) && result(key) : after(void* key) {}   (2)
```

Implementation. CCMOP's implementation is based on JavaMOP [8] and Clang [12]. We explain the two gray components in Figure 1 as follows.

- **Monitor Generator**. We reused the MOP syntax of JavaMOP and modified it to enable the usage of our AOP language in Figure 2 for defining events and the definitions of C/C++ code in event handlers. Besides, we have also developed the C++ runtime monitor code generator based on JavaMOP's RV-Monitor component. We support two specification languages: extended regular expression (ERE) and finite state machine (FSM). More specification languages in RV-Monitor are to be supported in the future.
- **Weaver**. We implemented the weaver for the AOP language in Figure 2 based on Clang. We find the pointcut matched statements by Clang's AST matching framework [11]. Besides, the instrumentation defined in the AOP

declarations is also carried out on AST, which is implemented by the AST transformation mechanism [1] of Clang. There are two advantages of AST-level instrumentation: First, compared with the source code text-based approach [10], it is more precise for the advanced mechanisms in C/C++ programs, such as `#define` and `typedef`, which are widely used in real-world programs; Second, AST-level instrumentation is carried out just before the IR generation, which enables just one time of parsing instead of two times needed by the source code text-based instrumentation method [10].

Limitations. There are following limitations of CCMOP. First, due to the widespread use of `typedef` in C/C++, some types are translated to other types on the AST-level. However, the description of ⟨Advice⟩ needs to specify the types in ASTs. For example, `std::string` is translated to `std::basic_string<char>` in ASTs, and we need to use `std::basic_string<char>` in ⟨Advice⟩ specification to capture the objects of `std::string`. Second, CCMOP's event specification is limited and only supports specific types of events, *e.g.*, method invocations, object constructions and memory operations. Third, CCMOP does not support multi-threaded C/C++ programs.

4 Evaluation

Basic Usage. CCMOP provides a script `wac` for compiling a single C/C++ file. The basic usage is demonstrated as follows, where we are compiling a single C++ file into the executable binary `demo`, and the RV with respect to the property will be carried out when running `demo`.

```
wac -cxx -mop <a property file> <a CPP file> -o demo
```

Besides, we also provide a meta-compiling [33] based script for real-world C/C++ projects with multiple files and employing standard build systems (*e.g.*, `make` and `cmake`). More details are provided on our tool's website[4].

We evaluate CCMOP for answering the following three research questions.

- **Applicability.** Can CCMOP support the RV of real-world C/C++ programs (especially C++ programs) with different scales?
- **Overhead.** How about the overhead of CCMOP when doing the RV of real-world C/C++ programs? Here, we only care about time overhead.
- **Soundness and Precision.** How about the soundness and precision of CCMOP? Here, soundness means the ability to detect all bugs, and precision means no false alarms.

[4] https://rv-ccmop.github.io

Benchmark Programs. To answer the first question, we applied CCMOP to different scaled benchmarks used in the literature of RV [10,34] and fuzzing [22]. Besides, we also get high-starred C++ projects from GitHub. Table 1 shows the benchmarks. In total, we have 100 real-world C/C++ programs. Our tool's website provides more details of our benchmark programs.

Table 1. C/C++ Benchmark Programs.

Type	Benchmark Name	Description
C	mini-benchmarks in MoveC [10,9]	126 mini-programs (2.6K LOCs in total)
	Ferry[34] and FuzzBench [22]	15 programs (2.5∼228.5K LOCs)
	High-starred GitHub Projects	35 programs (0.1∼239.5K LOCs)
C++	FuzzBench [22]	6 programs (10.5∼538.6K LOCs)
	High-starred GitHub Projects	44 programs (1.0∼675.2K LOCs)

Properties. Table 2 shows the properties used in the evaluation. The two properties, *i.e.*, use-after-free and memory leak, are used for both C and C++ programs. However, the event definitions are different. For C programs, we weave monitors when calling **malloc** and **free**; For C++ programs, we weave monitors to the **new** and **delete** statements.

Table 2. C/C++ Benchmark Properties.

Type	Property Name	Description
C	Use-after-free	Pointer is dereferenced after freed (`free`)
	Memory leak	Memory is allocated (`malloc`) but not freed
	Read-after-close	A `FILE` is read after close
C++	Use-after-free	Pointer is dereferenced after freed (`delete`)
	Memory leak	Memory is allocated (`new`) but not freed
	Safe Iterator	A collection should not be updated when it is being iterated

To answer the second question, we consider the C/C++ benchmarks that have many statements matching the property's pointcuts (*i.e.*, with non-negligible overhead) and provide test cases for demonstration and running. We compare CCMOP with LLVM's AddressSanitizer [27], which is widely used for memory checking of C/C++ programs, and the property is use-after-free[5].

To answer the third question, we applied CCMOP to SARD-100 [16] and Toyota ITC [29] benchmarks, in which source code is available. These two benchmarks focus on the detection of memory-related bugs (*e.g.*, memory leak and use-after-free). We evaluate CCMOP's soundness and precision for detecting memory leak

[5] We disable the other checkers in AddressSanitizer with options mentioned in website.

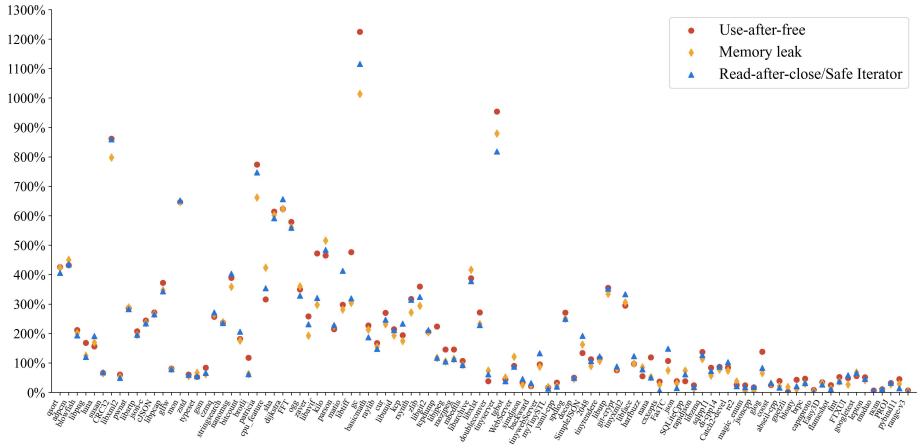

Fig. 3. Compilation time overhead.

and use-after-free bugs by running the programs with the test inputs provided by the benchmarks.

All the experiments were carried out on a laptop with a 2.60 GHz CPU and 32G memory, and the operating system is Ubuntu 20.04. The experimental result values of compilation time and runtime overhead are the averaged values of three runs' results.

Experimental Results. We applied CCMOP to do the RV for each benchmark program with respect to each property. Our tool can be successfully applied to 126 mini-benchmark programs from MoveC [10]. For the 100 real-world C/C++ programs, our tool can also support the weaving of RV monitors of all the properties during compilation. Figure 3 shows the result of compilation information.

The X-axis shows the program identities, where the first 50 programs are C programs, and the last 50 are C++ programs. The Y-axis shows the compilation time overhead compared with the original compilation time (denoted by C_O), *i.e.*, $(C_{RV} - C_O)/C_O$, where C_{RV} denotes the compilation time of RV. On average, the overheads of compilation time for use-after-free, memory leak, and read-after-close/safe-iterator are 2.01, 1.84, and 1.91, respectively. Due to the frequent usage of pointer operations, the overhead of use-after-free is usually the largest. For many programs (78%), the overhead is below 300%. There are nine small-scale programs whose overhead is over 500%. The reason is that the compilation of monitor code dominates the compilation procedure. For comparison, we applied LLVM's AddressSanitizer [27] to each benchmarks, and the average overhead of compilation time is 0.98.

Figure 4 shows the runtime overhead results for programs with notable changes.

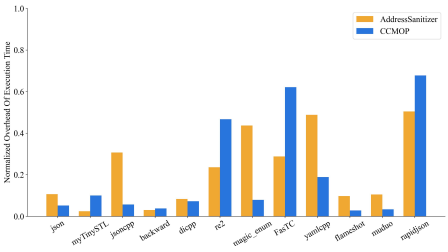

Fig. 4. Runtime overhead.

The X-axis displays the program name, and the Y-axis displays the overhead compared with the original program without RV. Each Y-axis value is calculated as follows: $log_{10}(T_{RV}/T_o)$, where T_o is the time of original program without RV, and T_{RV} is the time with RV. The figure shows that CCMOP's overhead is comparable with AddressSantinizer. The averaged overheads of AddressSantinizer and CCMOP are 81.9% and 88%, respectively. Furthermore, the results of the C benchmark programs (which have many memory operations) indicate that CCMOP's overhead is 961.6% on average. The detailed results are available on CCMOP's website.

Table 3 shows the results of soundness and precision on the two properties. For the property of memory leak, there are 5 and 18 test cases in SARD-100 [16] and Toyota ITC [29], respectively. CCMOP detected all bugs. For the property of use-after-free, there exist 9 and 15 test cases in SARD-100 and Toyota ITC, respectively. The majority of bugs (*i.e.*, 21 out of 24) can be detected. The reasons of missing bugs in three test cases are as follows: 1) in two test cases, there is no dereference of the freed pointer, but our property of use-after-free requires the dereference of the pointer; 2) in one case, there are nested pointer dereferences, on which CCMOP is limited and crashed. Besides, CCMOP does not produce any false alarms on these two benchmarks.

Table 3. Bug Detection Results of CCMOP.

Benchmark Name	Memory leak	Use-after-free
SARD-100 [16]	100%(5/5)	100%(9/9)
Toyota ITC [29]	100%(18/18)	80%(12/15)
Summary	100%(23/23)	87.5%(21/24)

5 Related Work

There already exist many RV tools developed in different backgrounds. Therefore, we divide the existing tools according to the implementation level of instrumentation.

Source-level. MoveC [10] is a RV tool for C programs and adopts source-code level instrumentation. MoveC supports the detection of segment, spatial and temporal memory errors, which is enabled by its monitoring data structure called smart status. Like MoveC, RTC [23] also implements the detection of memory errors and runtime type violations for C programs based on source-code instrumentation, and the implementation is based on the ROSE compiler platform [3]. E-ACSL [30] supports the checking of security properties for the C programs annotated with a formal specification language. Compared with these three tools, CCMOP supports the RV of C++ programs, and the instrumentation is carried out directly on ASTs. AspectC++ [5] is an AOP framework designed for C++

language and also instruments at the source-code level. However, AspectC++ does not support C programs well because the instrumented programs can only be compiled by a C++ compiler. Compared with AspectC++, our AOP framework has limited AOP features but supports both C and C++ programs.

IR-level. There exist some RV tools that instrument monitors at IR-level, including Google's sanitizers [27,32], SoftBoundCETS [25,24], and Memsafe[31], *etc*. These tools enjoy the benefits of IR-level instrumentation and support multiple languages. However, all of the mentioned tools can only support detecting memory-related properties. Besides these tools for C-family languages, we also classify JavaMOP [8] and Tracematches [4] into this category. Both tools adopt AOP-based instrumentation for runtime monitors, and the weaving is carried out directly on Java class files. These two tools inspire our tool, and our implementation is based on JavaMOP.

Binary-level. Some runtime monitoring tools adopt binary-level instrumentation of monitors. For example, MemCheck [28] is one of the most widely used tools for detecting memory errors and employs dynamic binary instrumentation (DBI) to implement memory runtime checks. Purify [2] is a commercial tool for detecting memory access-related errors for C/C++ programs and is also implemented based on DBI.

6 Conclusion and Future Work

This paper introduces CCMOP, a runtime verification tool for C/C++ programs. Inspired by JavaMOP [8], CCMOP adopts AOP-based monitor instrumentation and supports automatic monitor code generation and instrumentation. Moreover, CCMOP does instrumentation on the AST level. We have implemented CCMOP based on JavaMOP and Clang. To evaluate CCMOP, we have applied it to 100 real-world C/C++ programs, including 50 C++ programs. The experimental results indicate that CCMOP supports the RV of different scaled C/C++ programs and enables a transparent weaving of RV monitors.

The next step includes the following perspectives: 1) Improve the efficiency of RV by implementing more advanced RV optimization algorithms; 2) Support multi-threaded C/C++ programs; 3) Support more specification languages (*e.g.*, LTL and CFG).

Acknowledgments. This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62002107).

References

1. Clang: The Clang TreeTransform Class Template Reference, https://clang.llvm.org/doxygen/classclang_1_1TreeTransform.html
2. IBM: The Purify Documentation, https://www.ibm.com/support/pages/tools-purify
3. ROSE: Main Page, http://rosecompiler.org/ROSE_HTML_Reference/index.html
4. Allan, C., Avgustinov, P., Christensen, A.S.: Adding Trace Matching with Free Variables to AspectJ. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005. pp. 345–364. ACM (2005)
5. AspectC++: Aspectc++ Publications. https://www.aspectc.org/Publications.php (2021)
6. Bartocci, E., Deshmukh, J.V., Donzé, A.: Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In: Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 135–175. Springer (2018)
7. Chen, F., Meredith, P.O., Jin, D., Rosu, G.: Efficient Formalism-Independent Monitoring of Parametric Properties. In: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland. pp. 383–394. IEEE Computer Society (2009)
8. Chen, F., Rosu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005. Lecture Notes in Computer Science, vol. 3440, pp. 546–550. Springer (2005)
9. Chen, Z., Wang, C., Yan, J.: Runtime detection of memory errors with smart status. In: ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 296–308. ACM (2021)
10. Chen, Z., Yan, J., Kan, S., Qian, J., Xue, J.: Detecting Memory Errors at Runtime with Source-Level Instrumentation. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019. pp. 341–351. ACM (2019)
11. Clang: The AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html (2023)
12. Clang-15.02: Clang - A C language family frontend for LLVM. https://clang.llvm.org/ (2023)
13. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking. In: Proceedings of the NATO Advanced Study Institute on Deductive Program Design. pp. 305–349 (1996)
14. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model Checking and the State Explosion Problem. In: Tools for Practical Software Verification, LASER, International Summer School 2011. Lecture Notes in Computer Science, vol. 7682, pp. 1–30. Springer (2011)
15. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
16. Delaitre, A.: Test suite #100: C test suite for source code analyzer v2 - vulnerable (2015), https://samate.nist.gov/SRD/view.php?tsID=100
17. Google: sanitizers. https://github.com/google/sanitizers (2023)

18. Kiczales, G., Hilsdale, E., Hugunin, J.: An Overview of AspectJ. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference. Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer (2001)
19. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009)
20. LLVM: The LLVM Compiler Infrastructure Project. `https://llvm.org/` (2023)
21. Loveland, D.W.: Automated theorem proving: a logical basis
22. Metzman, J., Szekeres, L., Simon, L.: Fuzzbench: an open fuzzer benchmarking platform and service. In: ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1393–1403. ACM (2021)
23. Milewicz, R., Vanka, R., Tuck, J.: Runtime checking C programs. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. pp. 2107–2114. ACM (2015)
24. Nagarakatte, S., Zhao, J., Martin, M.M.K.: CETS: compiler enforced temporal safety for C. In: Proceedings of the 9th International Symposium on Memory Management, ISMM 2010. pp. 31–40. ACM (2010)
25. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009. pp. 245–258. ACM (2009)
26. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence. pp. 46–57. IEEE Computer Society (1977)
27. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker. In: 2012 USENIX Annual Technical Conference, Boston. pp. 309–318. USENIX Association (2012)
28. Seward, J., Nethercote, N.: Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In: Proceedings of the 2005 USENIX Annual Technical Conference. pp. 17–30. USENIX (2005)
29. Shiraishi, S., Mohan, V., Marimuthu, H.: Test suites for benchmarks of static analysis tools. In: 2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops. pp. 12–15. IEEE Computer Society (2015)
30. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In: RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Kalpa Publications in Computing, vol. 3, pp. 164–173. EasyChair (2017)
31. Simpson, M.S., Barua, R.: MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. Softw. Pract. Exp. **43**(1), 93–128 (2013)
32. Stepanov, E., Serebryany, K.: Memorysanitizer: fast detector of uninitialized memory use in C++. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015. pp. 46–55. IEEE Computer Society (2015)
33. travitch: The Whole Program LLVM Project. `https://github.com/travitch/whole-program-llvm` (2015)
34. Zhou, S., Yang, Z., Qiao, D.: Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In: 31st USENIX Security Symposium, USENIX Security 2022. pp. 4365–4382. USENIX Association (2022)