Concurrency defect localization in embedded systems using static code analysis: an evaluation

Bjarne Johansson[1,2], Alessandro V. Papadopoulos[2], Thomas Nolte[2]
[1] ABB Industrial Automation, Process Control Platform, Västerås, Sweden
[2] Mälardalen University, Västerås, Sweden
{bjarne.johansson}@se.abb.com,{alessandro.papadopoulos, thomas.nolte}@mdh.se

***Abstract*—Defects with low manifestation probability, such as concurrency defects, are difficult to find during testing. When such a defect manifests into an error, the low likelihood can make it time-consuming to reproduce the error and find the root cause. Static Code Analysis (SCA) tools have been used in the industry for decades, mostly for compliance checking towards guidelines such as MISRA. Today, these tools are capable of sophisticated data and execution flow analysis. Our work, presented in this paper, evaluates the feasibility of using SCA tools for concurrency defect detection and localization. Earlier research has categorized concurrency defects. We use this categorization and develop an object-oriented C++ based test suite containing defects from each category. Secondly, we use known and real defects in existing products' source code. With these two approaches, we perform the evaluation, using tools from some of the largest commercial actors in the field. Based on our results, we provide a discussion about how to use static code analysis tools for concurrency defect detection in complex embedded real-time systems.**

I. INTRODUCTION

Distributed Control Systems (DCSs) and the field device connectivity are the foundation in large scale automation applications, from automated harbor cranes and oil fields to paper mills and power plants. Unplanned stops in these domains can be costly, making the price tag for a defect causing downtime high. The DCS and fieldbus interfaces are typical hard real-time systems realized with embedded software running on automation industry suited hardware. The software sizes range from hundred thousands of Lines Of Code (LOC) to millions of LOC, often implemented as a multi-threaded application running on top of a preemptive Real-Time Operating System (RTOS), executed by a multi-core CPU.

Software defects are mistakes made in the software implementation that could manifest into an error, depending on if the execution and data flow expose the weakness. Concurrency defects are defects that manifest into an error due to an unintended interleaving pattern occurring in-between resources shared between two or more executing entities, such as threads or interrupts. Depending on the interleaving pattern required for the defect to manifest into an error, the manifestation probability can be low. Low probability defects can be hard to find in testing; they might never manifest into an error during the testing period. Once observed, a low reproducal rate can make it very time consuming to find the root cause, it can even take months [1]. The debugging constraint, implied by a real-time embedded system, compared to a Windows application, is likely to make it an even more time-consuming task.

This work is funded by The Knowledge Foundation (KKS), project ARRAY, and by The Swedish Foundation for Strategic Research (SSF).

Static Code Analysis (SCA) is the process for checking that source code adhere to industry standards and guidelines, such as MISRA[1]. SCA can be a manual code review, but more common is automated/tool-assisted SCA. When we refer to SCA, we refer to it as automated and tool-performed analysis. SCA tools of today are capable of execution and data flow analysis, together with the possibility of detecting potential concurrency defects. A weakness of SCA tools is that they do not have domain knowledge, implying that they need to act on general patterns, which increases the likelihood of false positives. A false positive is a defect indication where there is no real defect, and a false negative is a missing indication of a real defect. A low rate of false negatives means a high detection rate.

Contribution. First, this work provides, to the best of our knowledge, the first evaluation of the usefulness of SCA tools as a concurrency defect mitigation and localization approach in C++ implemented embedded real-time systems for the automation industry. Second, we provide a C++ based test suite that allows us to evaluate the SCA usability for different kinds of concurrency defects with various complexities. This test suite will enable us to analyze and present the usability per defect type and complexity level. The test suite is publicly available for download [2]. Finally, we present the evaluation result of the test on real, known, concurrency defects.

Outline. Section II presents an overview of related work. The tool selection and selection criteria are given in Section III and a concurrency bug categorization is presented in Section IV. Section V describes the test suites. The execution and result of the test are presented in Section VI where we also discuss about the result, and Section VII concludes the paper.

II. RELATED WORK

A significant amount of concurrency defect related research already exists. Even though the focus of this work is concurrency defects detection in the development phase by using SCA on the source code, it is important to emphasize that concurrency defects can be found and avoided in all stages of the development cycle: Design, implementation, and test execution. Model-based analysis can be used to detect design

[1]https://www.misra.org.uk

related concurrency defects [3], an overview is provided by Fu et al. [4]. There are many approaches to execution/dynamic concurrency bug detection, few examples being [5], [6], [7].

Earlier SCA research [8], [9] shows the value of SCA in general, but emphasizes that it is not the whole solution. Testing is still more efficient at finding defects. SCA tools need to act on general patterns since they do not have the domain knowledge, and therefore fail to detect domain-specific logic defects. Cost efficiency is another aspect. A tool must be able to find defects to earn its value. Wagner et al. [9] conclude that detection of three to four defects is what is needed for the evaluated tools to be cost-efficient.

Much SCA tool related research is performed using open source code with open source SCA tools. Arusoaie et al. [10] evaluate different open source SCA tools on a C/C++ test suite and find that Clang has the highest detection rate of concurrency bugs. The open source community [11] also uses SCA tools and for popular and advanced projects, SCA tool usage is already quite common.

Concurrency defect detection is an area that has received much research attention, especially on Java-based applications. Manzoor et al. [12] and Mamun et al. [13] evaluate how useful SCA tools are when it comes to finding concurrency defects in Java code. However, they come to very different conclusions; Manzoor et al. [12] conclude that none of the compared tools is better than the others, but Mamun et al. [13] conclude that the best tool has a detection rate of 48% and that the average detection rate of the compared tools is 25%. To reduce the number of false positives [14] propose using tools with different underlying techniques to maximize the detection rate. Voung et al. [15] uses RELAY on the Linux kernel. RELAY focuses on race conditions and produces a lot of false positives, that in turn are filtered through an unsound filter to reduce the amount. In the end, 53 race conditions where found. D'Silva et al. [16] conclude that concurrency defects is a problem for SCA tools.

Emanuelsson et al. [17] present a comparative study involving industrial tools and applications, including evaluations of the SCA tools performance and perception by the different teams. They compare Klocwork, Coverity, and PolySpace and categorize SCA in three categories, string and pattern matching, sound and unsound dataflow analyses. The only sound tool of the three tools is PolySpace, but the industrial users did not evaluate it. Another sound SCA tool is Astrée which uses abstract semantic on C code [18] to prove that a program is free from a defined set of security and run-time errors. It also includes a set of concurrency defects and it has been used in the industry [19], [20]. A strategy to handle a large amount of SCA tools gathered data from many teams with different needs in a large corporation is presented in [21].

Test suites are needed when evaluating SCA tools. Shiraishi et al. [22] provide a C, with a small fraction C++, test suite with test functions where specific defects are explicitly and intentionally created, together with a corresponding function that is OK and for which there shall be no violation report. The test suite is used for evaluation of a selection of SCA tools, where CodeSonar is found to be the best on detecting concurrency bugs. Arusoaie et al. [10] make some minor improvements of the same test suite and use it for evaluating SCA tools capabilities to detect vulnerability in C/C++.

To provide more realistic test suites, in terms of real, not injected defects, Lu et al. [23] use open source software with actual, known defects. They use it to evaluate dynamic bug finders (run-time detection), not SCA tools. Cifuentes et al. [24] present a test suite for C code bug detection with different evaluation aspects, such as scalability and accuracy. An extensive test suite is the Juliet test suite developed by the National Security Agency (NSA) Center for Assured Software (CAS) [25]. Juliet is made available through the National Institute for Standards and Technology (NIST) [26]. Juliet contains concurrency defects, such as data race. The data race defects in Juliet are implemented in C and do not expand over more than one file/compilation unit.

Earlier work presented in this section is about SCA in general or concurrency defect localization with SCA or other means, which is relevant for our work. However, no work targets concurrency defect mitigation using SCA tools in embedded C/C++ software with different degree of object-orientation. To the best of our knowledge, no such works exist.

III. Tool selection

Our tool selection methodology is fourfold. First, we interviewed senior professionals at ABB about which static code analysis tools they know and which they have been using. Then we turned to the internet; we used www.google.com with the search phrase "static code analysis c++". We analyzed the top results of that query and together with the output from the first step put together a list of 33 tools. As the third step, to reduce the number, we checked the product description for each of these tools to see if they fulfilled our selection criteria:

- C/C++ support.
- Windows support.
- Explicitly stated that the tool supports discovery of concurrency-related defects.

Astrée and seven other tools were ruled out for not having C++ support. Clang, which is the best open source tool for detecting concurrency defects [10], is not included since its concurrency detection is only available as an alpha, experimental, checker, for Unix/POSIX compatible target OS. Our RTOS is not POSIX compatible. PVS-Studio[2] and Eclair[3] had to be removed from the selection for not providing the possibility to adapt the analysis to the OS used. Coverity[4] had to be left out since the vendor did not wish to partake. Other tools, like PC-lint[5], were not considered in this study because of not having a product description claiming concurrency defect detection support.

The related work, see Section II, served as a fourth, sanity check, step of the selection, no additional tool were found.

[2]https://www.viva64.com/en/pvs-studio/

[3]https://www.bugseng.com/eclair

[4]https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

[5]https://gimpel.com/

Table I lists the resulting selection of tools. The list contains tools from GrammaTech, Parasoft and RougeWave, which are some of the largest SCA tool developers.

IV. Concurrency defect categorization

On a high level, concurrency defects are very similar, an accesses to a shared resource, or a lock, occur in an unintended way. On a more detailed level, the characteristics vary, which allows a break down into different defect categories [4], [31], [22] and [32].

We use the same categories as presented by Asadollah et al. in [32], which is also similar to what Shiraishi et al. present in [22] — summarized below. We use the term actor to denote an executing entity, such as a thread or an interrupt.

- Data race - unsynchronized simultaneous memory access from at least two actors, where at least one of them modifies the shared memory.
- Deadlock - a circular wait between at least two actors.
- Livelock - a circular wait, as the deadlock, with the differences that the actors are active during the wait, a.k.a. circular busy wait.
- Starvation - an actor does not get to execute since other actors consume all available CPU execution time.
- Blocking suspension - an actor waits for an unacceptably long time for a resource.
- Atomicity violation - overlapping execution of instructions modifying data result in different and possibly incorrect data value compared to if the data changing instruction execution would not overlap.
- Order violation - the intended order of data access is broken.

V. Test Suite

We developed a test suite for concurrency bug detection. It allowed us to test all different categories of concurrency bugs from Section IV, enabling us to evaluate the detection rate per category. In addition, we gathered ABB proprietary code bases with known concurrency defects from different products and versions, this allowed us to examine the usability against a real code base on actual, not injected, defects.

1) Concurrency defect test suite: Concerning concurrency defects, our test suite has a resemblance with the Juliet test suite [25]; both suites implement concurrency defects from the Common Weakness Enumeration (CWE)[6]. The difference is that our test suite is C++ based with an object-oriented design aimed specifically towards concurrency defects while Juliet covers a broad range of defects. Our test suite has higher complexity concerning function depth and class composition to increase the resemblance towards a complex embedded real-time application.

[6]https://cwe.mitre.org/

TABLE I
The selected tools

Product	Version	Vendor	Denoted	Ref.
CodeSonar	5.0.0	GrammaTech	T1	[27]
Klocwork	19.0.0.1278	RougeWave	T2	[28]
Parasoft C/C++ test	10.4.1	Parasoft	T3	[29]
Polyspace BugFinder	3.0	MathWorks	T4	[30]

The test suite contains an OS abstraction that allows us to evaluate if the tools can be used to detect concurrency defects on different RTOS. Our RTOS abstraction includes the most fundamental OS functions, such as necessary thread and semaphore handling. The test suite contains defects from each defect category in Section IV, except starvation. All tests have three basic actor combinations.

- Thread - thread $\mathrm{A_{tt}}$
- Thread - interrupt $\mathrm{A_{ti}}$
- Interrupt - interrupt $\mathrm{A_{ii}}$

In addition to the actor combinations, the test suite has different complexity and visibility levels. Example of varying complexity levels is the call depth from the actor, thread/interrupt entry function, to the defect and number of object instances passed on the way. Different visibility level examples are variation in the object instances invocation, achieved by using interfaces and polymorphism or concrete object instances. The combinations used are listed below.

- Method depth - three levels of method call depth until the error. One, three and seven levels deep. Denoted $\mathrm{MD_1}$, $\mathrm{MD_3}$ and $\mathrm{MD_7}$.
- Access visibility - three alternatives, the variables exposed to the concurrency defect can be a global variable directly accessed by the modifying method, a member variable directly accessed or a member reference to the shared variable. We denote the global variable access as $\mathrm{AC_g}$, the direct member variable access as $\mathrm{AC_d}$, and the access through a member reference as $\mathrm{AC_r}$. We use this complexity only for data race defects.
- Class composition depth - similar to the method depth, with the addition that the method calls are distributed over several classes. Denoted $\mathrm{CCD_0}$, $\mathrm{CCD_1}$, $\mathrm{CCD_3}$ and $\mathrm{CCD_7}$. $\mathrm{CCD_0}$ denote zero instance complexity, the defect is in the actor entry function. $\mathrm{CCD_{0s}}$ denote two actors that share the same entry function and $\mathrm{CCD_{0d}}$ denote that the two actors have two different entry functions. For both cases, the defect is in the entry function(s).
- Concrete class visibility - a complement to the $\mathrm{CCD_x}$ complexity with the addition that composition is either formed by the concrete classes, denoted $\mathrm{CCV_c}$ or with interfaces, implemented by the concrete class, denoted $\mathrm{CCV_i}$.

All defects have a correct counterpart, for false positive evaluation, except the data race with complexity $\mathrm{CCD_{0d}}$ and $\mathrm{CCD_{0s}}$ that only exist to check the detection of the simplest form of data race, similar to what is provided by Juliet [25].

Depending on the application, design and the implementors preference, mutual exclusion handling may have varying complexity and visibility degree. Our test suite contains two variants, from our experience, two commonly used practices. In our test suite, this serves the purpose of testing the capability to avoid false positives when the shared resource is accessed accurately with proper lock usage and also to test the detection of improper lock use, when using different lock acquisition patterns.

- Direct call, the protection is acquired directly with a lock acquisition call. Denoted $\mathrm{L_{dr}}$.
- Indirect call, using a resource acquisition is initialization (RAII) pattern. Denoted $\mathrm{L_{in}}$.

The number of defects in each category is listed in Table II.

2) ABB proprietary test suite: The ABB proprietary test suite consists of real-defects, not intentionally injected. Below we denote and describe each defect.

- $\mathrm{AD_1}$ is a deadlock defect involving two threads and two locks located in a C/C++ code base with a size in the range of millions of lines of code (LOC). Locks are retrieved through an OS abstraction where one of the involved locks are taken from an external component using a callback. Hence, the tool cannot pinpoint the exact location, but indicating that an external component is called from a critical section is desirable here. $\mathrm{A_{tt}CCD_7CCV_i}$ is a close approximation of the defect complexity using our taxonomy.
- $\mathrm{AD_2}$ is a data race, a read-modify-write, involving two threads. The defect location is a C/C++ code base in the range of millions LOC with a method depth of four from one thread and three from another and the execution path to the defect include function pointers and interfaces. $\mathrm{A_{tt}CCD_3AC_DCCV_i}$ is a close approximation of the defect complexity.
- $\mathrm{AD_3}$ is a data race, a read-modify-write of a hardware register, involving two threads. Located in a C code base in the range of a few hundred thousand LOC with a method depth of three, yielding a complexity and visibility close to $\mathrm{A_{tt}MD_3AC_G}$. The access is done using macros. To test if the detection depends on the access method, we created an alternative, where the hardware is accessed through a pointer directly, instead of using the access macros. We denote the macro version $\mathrm{AD_{3m}}$ and the pointer version $\mathrm{AD_{3p}}$.

The complexity of these defects, especially $\mathrm{AD_1}$ and $\mathrm{AD_2}$ is high. Making them a suitable counterpart to the concurrency defect test suite, that has different complexity levels.

VI. Execution and result

A. Execution

We installed the tools on a virtual machine running Windows 10, and each tool was configured and adapted for the toolchains used and also for the task at hand; concurrency defect detection in our concurrency test suite and of the known defects in the ABB proprietary code. The configuration and rules are available for download at [2]. Typically the rules for concurrency defect rules had to be enabled and the rules not related to concurrency defects were disabled. Configurations were needed to make the tool understand the OS abstraction used. We did not utilize the option to make customized rules that some tools provide.

Besides from rule enabling/disabling and OS adaptations, T1 provide possibilities to adapt the analysis depth with the help of, for example, time limits for different analysis phases. Increasing the limits increase the time for the analysis. The individual execution time for the test suite analysis is below one hour for each test suite for all tools, including T1 with default depth. However, with the increased depth, that time went up to 33 days for the concurrency test suite.

To present the result from the execution we use the statistics proposed by Shiraishi et al. in [22], summarized below.

$$DR = \frac{DV}{AV} \times 100$$

DR is the detection rate, DV is the number of detected violations and AV is the actual number of violations in the test suite.

$$FPR = \frac{RFP}{NV} \times 100$$

FPR is the false positive rate and RFP is the number of reported false positives and NV is the number of non-violations, "violation provocations".

B. Result

In the two subsections below we present the result from the evaluation using the concurrency defect test suite and the ABB proprietary code based test suite.

1) Concurrency defect test suite: Table II show DR and Table III show the FPR per defect category. True and false positives that are reported on lines that are not in the test scope are not considered.

In addition to presenting the result of the different defect categories, we summarize the result for the various complexities. The statements below are valid for all tested tools unless stated otherwise. When we say that the result is the same, we mean the different complexity levels of the specified complexity, not that the result is the same between various tools.

- Actor combination - T2, T3 and T4 showed no differences in reported violations between different actor combinations. The same violation were detected for $\mathrm{A_{tt}}$, $\mathrm{A_{ti}}$ and $\mathrm{A_{ii}}$. T1 found no data races from actor $\mathrm{A_{ii}}$. We modeled interrupts as signals in tool T1, if we would model interrupts as threads instead of signals, we expect the same result for actor $\mathrm{A_{ii}}$ as for actor $\mathrm{A_{tt}}$.
- Method depth - no difference in the result between different method depths. The same violations were detected for $\mathrm{MD_1}$, $\mathrm{MD_3}$ and $\mathrm{MD_7}$.

TABLE II
Detection rate on the concurrency test suite. Per defect category.

Defect category	Total AV	Tools			
		T1	T2	T3	T4
Data race	89	6%	0%	0%	24%
Deadlock	108	19%	50%	0%	0%
Livelock	54	100%	100%	0%	0%
Blocking suspension	27	0%	0%	0%	0%
Atomicity violation	108	0%	0%	50%	0%
Order Violation	27	0%	0%	0%	0%

- Access visibility - the result differed between AC complexities, defect and tools. T2 and T3 did not detect any data race violation. T4 detected data race with complexity AC_g, but no violation with complexity AC_d or AC_r. T1 detected data race with complexity AC_g and gave false positives violation for AC_d and AC_r member variables protected with L_{in}, the RAII guards, but raised no violation for the unprotected AC_d and AC_r member variables.
- Class composition depth - T2, T3 and T4 detected violations and were unaffected of the depth CCD_0, CCD_{0s}, CCD_1, CCD_3 and CCD_7. T1 found data race violation for depth CCD_0 and CCD_{0s} but not for deeper depths.
- Concrete class visibility - none of the tools detected true positive data race violations with a CCV_i complexity. Only data race defects created by CCV_c composition were detected. T1 and T2 found potential deadlocks regardless if the composition was made by interface or concrete classes. T1 gave a false positive data race violation for CCD_1CCV_i, hence, it was able to trace over one interface but misunderstood the RAII pattern that protected the member variable, hence the false positive.
- Direct/indirect call for lock acquisition - none of the tools handled the L_{in} complexity. For T1 and T4, 100% of the false positives are due to RAII guard recognition failure.

2) ABB proprietary test suite: Table IV shows the detection per defect and tool. After evaluating the tools on the concurrency test suite, we learned that the complexity of the ABB defects is higher than handled by the tools. We denote this as too high complexity (THC) in Table IV.

C. Discussion

The result shows that the detection of concurrency related defects is possible but is very much tool and complexity dependent. Data race defects are only detected by T1 and T4

TABLE III
FALSE POSITIVE RATE ON THE CONCURRENCY TEST SUITE. PER DEFECT CATEGORY.

Defect category	Total non-defects	Tools			
		T1	T2	T3	T4
Data race	162	9%	0%	0%	15%
Deadlock	108	0%	0%	0%	0%
Livelock	54	0%	0%	0%	0%
Blocking suspension	27	0%	0%	0%	0%
Atomicity violation	108	0%	0%	0%	0%
Order Violation	27	0%	0%	0%	0%

TABLE IV
DEFECT DETECTION IN ABB PROPRIETARY TEST SUITE. YES, DEFECT DETECTED. NO, DEFECT NOT DETECTED. THC, DEFECT COMPLEXITY TOO HIGH FOR THE TOOL.

Defect	Tool			
	T1	T2	T3	T4
AD_1	THC	THC	THC	THC
AD_2	THC	THC	THC	THC
AD_{3m}	No	THC	THC	No
AD_{3p}	Yes	THC	THC	No

and are very complexity dependent. T4 was only able to detect data race in AC_g, global variables, not member variables. Neither T1 nor T4 were able to understand CCV_i and thereby not able to trace execution flow from the function entry to the data race defect over interface use. The feasibility of using SCA tools for data race detection depends on the code base, lesser use of interfaces, the more analysis coverage, and a higher likelihood of finding potential data races.

T1 and T2 found potential deadlock situations and they did so regardless of CCV complexity. However, they failed to recognize L_{in} and therefore failed to recognize deadlocks created using RAII guards. All the false positive data races reported by T1 is a consequence of not interpreting the RAII guards correctly.

From the result of the ABB test suite, we see that the tools did not support the complexity in AD_1 and AD_2. For AD_1, there was no default rule to check if an external component, with unknown implementation, is called from within a critical section. For AD_2 the problem is the CCV_i complexity, the execution from the thread entry to the defect includes interfaces, with deeper depth than any tool managed in the concurrency test suite. Only T1 managed to detect a AD_3 defect, the AD_{3p}. No tool found the AD_{3m} defect.

Another aspect to consider is the analysis time. T1 required close to a month of execution time when using long timeouts and extensive depth setting, to avoid false negatives due to shallow analysis. That long execution time is not feasible if used as a traditional SCA tool when the analysis must be executed before each source code change commit. We propose concurrency related SCA to be incorporated as a step in a products' development cycle, executed with appropriate intervals, similar to time-consuming function and product tests. T1 analysis time can be significantly decreased, compared to our times, by using more computational power.

Analysis of false positives require man hours and therefore should be kept to a minimum. We believe that the steps below can be a practical way to handle the concurrency SCA. We assume that traditional, compliance checking, SCA is already a part of the process.

- The concurrency analysis should only have the relevant rules enabled, the rules that require long analysis time analysis such as data race. The other rules, that are not analysis-time consuming, should be checked before every check-in, as part of the regular SCA use.
- Establish a baseline after the first run in which all the violations are handled and the false positives suppressed.
- Run as frequently as possible. Frequent runs keep the number of new false positives low, and the real defects short lived.

D. Threats to validity

An obvious threat is the selection of tools, not included/found tools may be significant diverse at concurrency defect detection. Another threat to validity is the test suite. C++ allows a programmer to construct source code in various forms, and a test suite can not cover all of them. To reduce this threat, we used well-known and general C++ patterns with different levels of complexity, to implement concurrency defects from each of category presented by Asadollah et. al in [32], except starvation. However, the constructs in the test suite are likely simpler than in real-software and do not cover all construction possible. Hence, the detection rates we got are likely higher than for real software.

VII. Conclusion

By developing a C/C++ based test suite, we have been able to evaluate the feasibility of SCA tool usage for concurrency defect detection in object-oriented code bases for embedded real-time products. We have shown that the usefulness of SCA for concurrency defect detection depends on the code base and defect category. Data races are more likely to be found in a code base where object composition consists of concrete classes instead of interfaces. Data race in global variables is also more likely to be found compared to data race in member variables. In other words, data races are more likely to be found in a more procedural code base than in an object-oriented code base. The success rate of lock related defect detection, such as deadlock and livelock detection, is depending on how the locks are acquired, rather than the execution path to the critical region. No tool was able to recognize the RAII pattern. None of the tools detected order violation nor blocking suspension defects. The analysis time for some tools is long. We envision that using SCA for concurrency defect detection is not something that is executed for every commit/push, but instead with intervals that are suitable for the product development and release cycle and the execution time of the SCA for the particular product.

VIII. Future work

Potential candidates, such as Axivion Bauhaus Suite [7] and LDRA tool suite [8], not included in this work, could be part of a more extensive future study.

A test suite for SCA testing can always be improved and extended with more complexity and different patterns. Some of the more immediate improvements include using different semaphores in the different dead and live lock tests for the RAII guards and the native OS call and by adding support for and use of C++11 and C++17 threads and lock mechanism. Due to the complexity and variety of configuration settings provided by the tools, it is hard to know if the used parameter settings are optimal for the purpose. Future work could look deeper into configuration and the possibility to create own, purpose-fit rules, that most of the tools provide. In Section VI-C we introduce one way to incorporate concurrency defect mitigation with SCA in the development process. An evaluation of this process step included in the development cycle for a real product is a natural continuation together with a cost evaluation. We looked on three real defects from large code bases of complex products. Future work could assess SCA for concurrency defects on resource-constrained devices since the structure of such products' source code might be different and to some extent simpler and more procedural oriented.

[7]https://www.axivion.com

[8]https://ldra.com/

References

[1] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: An exploratory survey," in *CAV ws Expl. Conc. Efficiently and Correctly*, 2008.

[2] "Concurrency test suite and tool configuration." https://github.com/Burne77a/ConcurrencySCATestSuite. Accessed: 2019-06-26.

[3] M. Shousha, Y. Labiche, and L. C. Briand, "A UML/MARTE model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems," *IEEE Tr.Soft.Eng.*, vol. 38, 2010.

[4] H. Fu, Z. Wang, X. Chen, and X. Fan, "A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques," *Software Quality Journal*, vol. 26, no. 3, pp. 855–889, 2018.

[5] W. Wang, Z. Wang, C. Wu, P.-C. Yew, X. Shen, X. Yuan, J. Li, X. Feng, and Y. Guan, "Localization of concurrency bugs using shared memory access pairs," in *29th ACM/IEEE ASE*, pp. 611–622, 2014.

[6] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *SIGPLAN Not.*, vol. 44, no. 6, pp. 110–120, 2009.

[7] S. A. Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "A runtime verification tool for detecting concurrency bugs in freertos embedded software," in *17th ISPDC*, pp. 172–179, 2018.

[8] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Soft. Eng.*, vol. 32, no. 4, pp. 240–253, 2006.

[9] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for Java," in *1st Int. Conf. on Software Testing, Verification, and Validation*, pp. 248–257, 2008.

[10] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in C/C++ code," in *19th SYNASC*, pp. 161–168, 2017.

[11] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd SANER*, vol. 1, pp. 470–481, 2016.

[12] N. Manzoor, H. Munir, and M. Moayyed, "Comparison of static analysis tools for finding concurrency bugs," in *IEEE 23rd ISSRE Wksp*, 2012.

[13] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for Java concurrency bugs," in *Third Swedish W. on Multi-Core Computing (MCC-10)*, 2010.

[14] D. Kester, M. Mwebesa, and J. S. Bradbury, "How good is static analysis at finding concurrency bugs?," in *10th IEEE SCAM*, pp. 115–124, 2010.

[15] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *ESEC-FSE '07*, pp. 205–214, 2007.

[16] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Tr.Comp.-Aided Design of Integrated Circuits and Sys.*, vol. 27, no. 7, 2008.

[17] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *E.Notes Theor. Comput. Sci.*, vol. 217, pp. 5–21, 2008.

[18] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *4th ACM SIGPLAN-SIGACT POPL*, pp. 238–252, 1977.

[19] D. Kästner, L. Mauborgne, and C. Ferdinand, "Detecting safety-and security-relevant programming defects by sound static analysis," in *2nd Int. Conf. on Cyber-Technologies and Cyber-Systems*, vol. 2, 2017.

[20] A. Miné and D. Delmas, "Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software," in *12th Int. Conf. on Emb. Soft.*, EMSOFT '15, pp. 65–74, 2015.

[21] A. Dubey, K. Saha, and J. Hudepohl, "Reporting and assessment of static analysis policies in a globally distributed organization," in *IEEE 9th Int. Conf. on Global Software Engineering*, pp. 84–89, 2014.

[22] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *IEEE ISSREW*, pp. 12–15, 2015.

[23] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In W. on the Evaluation of Software Defect Detection Tools*, 2005.

[24] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "BegBunch: Benchmarking for C bug detection tools," in *2nd DEFECTS '09*, pp. 16–20, 2009.

[25] "Static analysis tool study methodology," Center for Assured Software (CSA), NSA, 2011.

[26] "NIST. national institute of standards and technology samate reference dataset (srd) project." https://samate.nist.gov/SRD. Access: 2019-03-06.

[27] "Codesonar." https://www.grammatech.com/products/codesonar. Accessed: 2019-04-12.

[28] "Klocwork." https://www.roguewave.com/products-services/klocwork. Accessed: 2019-04-12.

[29] "Parasoft C/C++ test." https://www.parasoft.com/ctest/static-analysis. Accessed: 2019-04-12.

[30] "PolySpace bug finder." https://se.mathworks.com/products/polyspace.html. Accessed: 2019-03-06.

[31] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Emp.Sw.Eng.*, vol. 19, no. 6, 2014.

[32] S. A. Asadollah, H. Hansson, D. Sundmark, and S. Eldh, "Towards classification of concurrency bugs based on observable properties," in *COUFLESS '15*, pp. 41–47, 2015.