International Journal of Engineering & Technology, 7 (2.21) (2018) 454-457

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

An efficient software verification using multi-layered software verification tool

S.V. Gayetri Devi[1*], C. Nalini[2], N. Kumar[3]

[1]*Department of Computer Science & Engineering, Bharath Institute of Higher Education and Research, Bharath University, Chennai.*
[2]*Department of Computer Science & Engineering, Bharath Institute of Higher Education and Research, Bharath University, Chennai.*
[3]*Department of Computer Science & Engineering, Vels Institute of Science, Technology & Advanced Studies(VISTAS), Chennai, India.*
**Corresponding author E-mail:gayetri.venkhatraman@gmail.com*

Abstract

Rapid advancements in Software Verification and Validation have been critical in the wide development of tools and techniques to identify potential Concurrent bugs and hence verify the software correctness. A concurrent program has multiple processes and shared objects. Each process is a sequential program and they use the shared objects for communication for completion of a task. The primary objective of this survey is retrospective review of different tools and methods used for the verification of real-time concurrent software. This paper describes the proposed tool 'F-JAVA' for multithreaded Java codebases in contrast with existing 'FRAMA-C' platform, which is dedicated to real-time concurrent C software analysis. The proposed system is comprised of three layers, namely Programming rules generation stage, Verification stage with Particle Swarm Optimization (PSO) algorithm, and Performance measurement stage. It aims to address some of the challenges in the verification process such as larger programs, long execution times, and false alarms or bugs, and platform independent code verification

Index Terms: *Verification, concurrency, contracts, swarm optimization.*

1. Introduction

Verification and Validation are vital actions in the development of software products [3]. Verification ensures that the software is aligned with its desired requirements, meets particular necessities, completeness, and performance according to specifications [4]. It strives to fulfill consistency, the correctness of program translations as well as behavioral correctness [6]. Validation is the determination of the correctness of the software at every stage of the development cycle [5].

The two categories of software verification are Static analysis and Dynamic analysis. Static analysis is the verification of artifacts in software and locating abundant defects using templates. This analysis is highly automated and completely relies on testing tools, which can be applied only to source code and some specific artifacts [18]. Dynamic analysis performs software system characteristic analysis and evaluation based on its real efforts of prototypes of the system. Examples include simulation testing, monitoring, and profiling [18].

Methods of verifying software can be classified as inspection, testing, analysis, and demonstration. Inspection is defined as the examination of a software without any destruction using one of the five senses (Visual, auditory, olfactory, tactile and taste). It may include some physical manipulation and measurements. The common techniques involved in the inspection are desk checking, walkthroughs, software reviews, technical reviews, and formal inspections (e.g., Fagan approach) [20]. Testing is defined as verification of a software using the predefined sequence of inputs, data or stimuli to make sure the requirements for the execution of specific and predefined output. It can be performed by various techniques such as statement coverage, condition coverage, and decision coverage.

Analysis of a software system involves prediction of product failure using non-destructive tests to hypothesize the breaking point. Demonstration of a software is also known as input/output driven testing, in which the derived outputs are compared against the expected output values. The most accepted technique of demonstration is error guessing, boundary-value analysis, and equivalence partitioning [20].

2. Literature review of different verification tools for concurrent programs

This segment presents a detailed literature survey on the Static and Dynamic software verification. It gives the comparative description of various methodologies and tools that are utilized in software verification.

The contracts for accessing the public modules can be verified and validated by two methods, namely static and dynamic while giving real-time errors. These methods of verification not only identify the violations in atomicity but its order in a contract. From the dynamic approach, it is easy to support contracts along with the agreements and spoilers (a set of sequences of methods that may violate a target) [7] [8] constructed a tool CONC2SEQ for code transformation, which is a FRAMA-C plugin. This tool transforms the concurrent C code and generates a sequential code with multithreading simulated by interleavings. From this approach, it seems to be automatic code generation with user specifications incorporated again without any manual involvements. [9] reduced the interleaving instances through program instances generated by

code to code translations. This approach consumes less time and memory dimensions of the backend analysis tool (Lazy-Cseq).

[10] derived two approaches for the large and real case of C-codebases, they are CBMC and FRAMA-C. The author commented about their efficiency stating FRAMA-C is not readable, whereas the output generated by CBMC has been richer since FRAMA-C contains pointers information and missing memory allocations. He also described the SonarQube, which is a platform that supports code inspection activities. [11] developed lazy sequentialization for the multi-threaded programs partial store order (PSO) and total store order (TSO). This prototype tool is very much effective and competitive with current tools on standard benchmarks that are used before.

[12] reported about VerCors tool which supports the feature of concurrency such as kernel using barriers, heterogenicity, atomic operations, and compiler directives. VerCor tool has a characteristic feature that it can generalize the concurrent program verifications to a language-independent setting where a front end can be added easily. [13] developed a tool named 'Atomchase' a new technique that directs the execution of a dynamic analysis tool towards three-access pattern (TAP) instances. Using 27 benchmarks comprising 5.4 million lines of Java, they compared AtomChase to five other tools. AtomChase found 20% more TAP instances than all five tools combined.

[17] presented an approach to help the developers in verifying if the work units, which have triggered bugs due to certain violations of atomicity. The units are verified if they are sufficiently synchronized or not by locks introduced for fixing the bugs. This approach effectively verifies the synchronizations by testing only a minimal set of suspicious atomicity violations without any prior knowledge of the work units. Thus, it becomes more practical and efficient than other approaches.

Table I: Comparative Study on Different Verification Models in Concurrent Software

Paper & Author	Verification method	Description	Parameters measured/Benchmarks	Merits	Limitations
Verifying Concurrent Programs Using Contracts Dias et.al., (2017).	1. Generate Contracts by extracting from source code, libraries or software modules 2. Extending Contracts with Parameters 3. Extending Contracts with Spoilers 4. Static and Dynamic contract validation	The input opted is Multi-threaded C/C++ program	Number of Clauses of contracts, Contract Violations, False Positives, Potential AV, Real AV, Number lines of code SLOC and the time of completion of analysis	It not only detects the atomicity violations but also the order violations in a contract	1. The contracts (sequence of methods) for verifications are derived by developers. 2. Cost enough in practical applications
CONC2SEQ: A FRAMA-C Plugin for Verification of Parallel Compositions of C Programs Blanchard et.al., (2016).	The CONC2SEQ plugin transforms the original code into a sequential code to stimulate the concurrent behavior of the program which can be performed in FRAMA-C.	The input applied is concurrent C-program	1. 704 obligations are automatically proved with Frama-C Aluminium using Alt-Ergo 1.01 and Z3 4.4.2. 2. It takes 260s on a QuadCore Intel Core i7-4800QM @2.7GHz.	Automatic integration of user specifications into the new code without any manual interventions	Some of the constructs like value analysis and runtime verification plugins are not handled by the FRAMA-C plugins
Parallel bug-Finding in concurrent programs via reduced interleaving instances Nguyen et.al., (2017).	1. Chain of code to code transformation of input program into set of simpler program variants by implementing VERISMART tool based on Cseq framework 2. Analysis of each program variant by selecting an input tiling and identify potential software bug. - Lazy-Cseq, symbolic analysis tool.	Concurrent C programs that use the concurrency library POSIX threads.	1. Concurrency Benchmarks namely eliminationstack and safestack under SC and PSO memory models exposed reporting the minimum, the maximum, the average and the standard. 2. Deviation over the verification Time (in seconds) and Memory (in MB) consumption of buggy variants and Percentage of Instances with bugs.	Consumes less time and memory dimensions	Not able to find the bug for safestack-TSO
Bounded Model Checking and Abstract Interpretation of Large C Codebases Martignano (2017).	Execution of bounded model checking (via CBMC) and abstract interpretation (via Frama-C) 1. Generation of a model of the code under analysis 2. "Symbolic execution" or "logic verification" of the model itself.	Large, real case, C-codebases are used as input.	Clang Static Analyzer has been integrated (via Clang-Tidy) into SonarQube C/C++ Community Plugin.	Since the code itself able to see the results it substantially facilitate certain activities like software verification and validation, quality assessment, and bug findings.	CBMC cannot be so easily integrated in the standard build chain as Clang Static Analyzer or Facebook Infer.
Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions Tomasco et.al., (2016).	Developed lazy sequentialization for the multi-threaded programs partial store order (PSO) and total store order (TSO).	Input: C programs with POSIX threads in a prototype tool called LazySMA.1	Abstract data type that factors out the semantics of the memory model, allowing us to reuse tools designed for the analysis of concurrent programs under SC.	Much effective and competitive with current tools on standard benchmarks that are used before.	Cannot be extended to further WMMs.
The VerCors Tool Set: Verification of Parallel and Concurrent Software Blom et.al., (2017).	Captures the behavior of a shared memory concurrent program by means of a process algebra term with data.	Input: C-program	Verifying functional correctness of three different concurrency features: heterogeneous concurrency, kernels using barriers and atomic operations, and compiler directives for parallelization.	It can generalizes the concurrent program verifications to a language independent setting where a front ends can be added easily	Less scalability
Efficient Detection and Validation of Atomicity Violations in Concurrent Programs Eslamimehr et.al., (2017).	The execution of a dynamic analysis tool towards three-access pattern (TAP) instances. 1. Static analysis approach 2. Dynamic approach 3. Dynamic predictive approach	5.4 million lines of Java code	AtomChase found 20% more TAP instances than all five tools combined.	Efficient and Accurate Verification	1. HAVE to produce TAP candidates 2. This approach relies on a constraint solver both in the plan synthesis and directed execution modules. 3. AtomChase cannot filter benign atomicity violations.

					4. This approach has no support for native code.
Formal Verification With Frama-C: A Case Study in the Space Software Domain e silva et.al., (2015).	Explores abstract interpretation and deductive verification by employing Frama-C's value analysis and Jessie plug-ins to verify embedded aerospace control software.	Source codes	Both approaches can be employed in a software verification process to make software more reliable.	Analyzed results from Frama-C's value analysis plug-in indicate that the algorithms are correctly implemented without problems such as division by zero, invalid pointer access, buffer overflows, and other run-time errors.	The manual generation of contracts that are written as Annotations in Source Code, become very tedious to generate for larger program size. The paper focuses on Formal verification of C Software which is platform dependent. Contains a large number of floating-point computations.
Towards Deductive Verification of Concurrent Linux Kernel Code with Jessie Mandrykin et.al., (2015)	Verification of concurrent code working with shared data by proving the code's compliance with specified synchronization discipline.	Linux Kernel	Both approaches can be employed in a software verification process to make software more reliable. Initial functional specification for the exposed interface of the RCU synchronization mechanism to show how the VCC ownership methodology (with a minor extension) can be applied for verification of Linux kernel modules.	No task on inherent possible source of unsoundness of the ownership methodology arising from possible unrestricted usage of atomic blocks.	It has not been formally verified using model-checking or some other suitable technique.
Debugging Multithreaded Programs Using Symbolic Analysis Xiaodong Zhang (2017).	Guided Execution Symbolic Analysis Branch Scanning	Multithreaded C programs	1. The bug detection capability of Proactive Debugger tool is compared against two concurrent software testing tools – ESBMC and Maple. 2. Proactive-Debugger detects the bugs in all the experiments. 3. Study conducted on eleven benchmarks - account, arithmetic, queue, Stack, FFT, lu continous, lu non continous, radix, pfscan, and swarm.	The net effect of applying this feedback loop is a systematic and complete coverage of the program behavior under a fixed test input.	Very tedious to generate bugs for larger program size. The work does not address Dynamic Analysis that must complement. Formal verification as part of V&V process to ensure complete Software Reliability.
Verifying Synchronization for Atomicity Violation Fixing Shi et al., (2016)	1. Every program execution is modeled as a trace of events, which must obey some basic constraints such as the data/control flow of the program and the synchronization semantics. 2. Combining the fortes of both bug-driven and change-aware techniques, which enables SWAN to effectively verify synchronization.	Java programs.	Insufficient synchronization was avoided by testing a minimal set of cautious violations.	Insufficient Synchronizations is time consuming. Pervasive in real world programs. Verification algorithm can converge much faster.	Insufficient synchronizations are common and difficult to be found in software development.

3. Research findings

In software verification, the contracts derived by developers for the verification of concurrent programs are time-consuming, and hence not cost-effective for practical applications [7], and are platform dependent [14]. Verification of parallel programs using FRAMA-C plugins shows inefficient results in value analysis and runtime error detection [8]. Reducing the interleaving instances during bug detection is inapplicable in safestack TSO [9]. As in clang static analyzer and Facebook infer, it is not possible to integrate with standard chain for bounded model checking of C-codebases through CBMC and abstract interpretation [10]. Lazy sequentialization for TSO and PSO is not applicable for the weak memory models [11]. There was a loss in scalability during the verification of concurrent C-programs by VerCors tool and also it was inferred with low detection of bugs by this tool [12]. Atomchase a tool proposed by [13], can recognize about 89% of the real atomicity violations, but it is unable to find and filter the benign atomicity violations. Similarly, SWAN tool shows insufficient synchronization when detecting atomicity violations in synchronizations [17]. Debugging by symbolic analysis is laborious for larger programs [16]. Deductive verification of concurrent Linux Kernel code by model checking is not been formally verified [15].

4. Proposed system for verification of concurrent Software

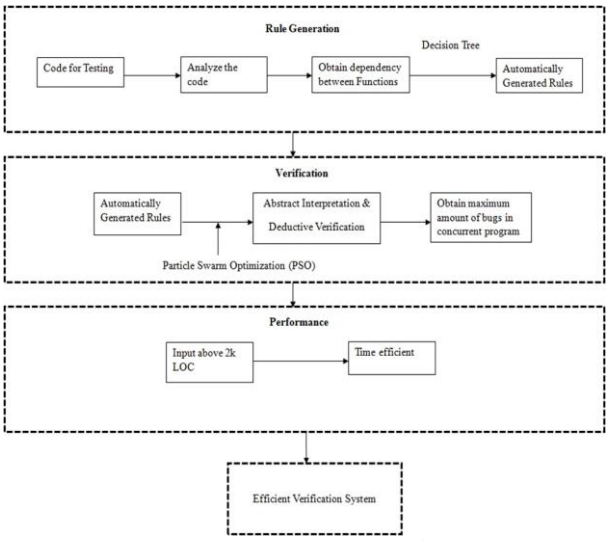

Figure 1: The proposed multilayered verification system

The proposed working line provides a framework to mitigate various verification issues in Formal methods such as handling large-sized programs, strenuous generation of programming contracts by developers, and platform dependency. FRAMA-C is unable to handle value analysis plugin because it does not identify

the function termination and shows valid results even though the input data is incorrect. To overcome these issues, our proposed system provides a multilayered verification tool named F-Java for concurrent Java programs. The framework consists of Programming rules generation stage, Verification stage with Particle Swarm Optimization (PSO) algorithm, and Performance measurement stage. In this tool, contracts are generated in order to ensure the atomicity of the given input. They provide an effective mechanism to establish the difference between the components to be verified and specifies the Abstract interpretation and Deductive verification made by the F-Java tool. The deadlocks and atomicity in the contracts are then evaluated by the deadlock detection (DD) algorithm. After the deadlock detection, the functional dependency between the set of attributes in methods and classes (e.g., Student_Id->Student_Name.) in the logical sequences are analyzed. The rules to be verified are automatically generated in the form of a decision tree. Now the contracts and the data to be verified are optimized by the particle swarm optimization (PSO). This optimization aims at the efficient prioritization of the contracts and test cases in a Time constrained verification environment.

An Abstract interpretation technique is used to identify non-functional errors such as timing, memory usage and hence ensure the absence of runtime errors. Deductive verification is used instead of the common formal verification method. Because, formal method of verification shows specification errors, incomplete functional coverage of specification, and bugs can miss design errors, which are the limitations that destructs the concurrent software. Deductive verification deducts bugs such as buffer overflows, run time-errors, undefined functions and obfuscate codes with infinite state. The verification tool thus aims at maximum number of bugs being detected with time efficiency and increasing performance.

5. Conclusion

Various research papers on verification of concurrent programs were studied and with a comparison of their models and drawbacks briefly discussed. Based on the research findings, the verification of concurrent Java programs by using F-Java tool is proposed. This system encompasses, Abstract interpretation, Deductive verification and Optimization algorithms, to verify the multi-threaded Java programs efficiently with a reasonably lesser execution time. The future working line, thus paves the way for the new possible research directions in software verification.

References

[1] Philippaerts P, Mühlberg JT, Penninckx W, Smans J, Jacobs B & Piessens F, "Software verification with VeriFast: Industrial case studies", *Science of Computer Programming*, Vol. 82, (2014), pp.77-97.

[2] Schmidt RF, "Software engineering architecture-driven software development", *Amsterdam: Elsevier,* (2013)

[3] Monteiro P, Machado RJ & Kazman R, "Inception of software validation and verification practices within CMMI Level 2", *Software Engineering Advances,* (2009), pp.536-541.

[4] Filliâtre JC, "Deductive software verification", *International Journal on Software Tools for Technology Transfer (STTT)*, Vol.13, (2011), pp.397-403.

[5] Knutson C & Carmichael S, "Verification and Validation", *Embedded Systems Programming*, Vol. 25, (2001).

[6] Gabmeyer S, Kaufmann P, Seidl M, Gogolla M & Kappel G, "A feature-based classification of formal verification techniques for software models", *Software & Systems Modeling*, (2017), pp.1-26.

[7] Dias RJ, Ferreira C, Fiedor J, Lourenço JM, Smrcka A, Sousa DG & Vojnar T, "Verifying Concurrent Programs Using Contracts", *Software Testing, Verification and Validation (ICST)*, (2017), pp.196-206.

[8] Blanchard A, Kosmatov N, Lemerre M & Loulergue F, "Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs", *Source Code Analysis and Manipulation (SCAM)*, (2016), 767-772.

[9] Nguyen TL, Schrammel P, Fischer B, La Torre S & Parlato G, 'Parallel bug-finding in concurrent programs via reduced interleaving instances", *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, (2017), pp.753-764.

[10] Martignano M, "Bounded model checking and abstract interpretation of large C codebases", *Metrology for AeroSpace (MetroAeroSpace)*, (2017), pp.16-20.

[11] Tomasco E, Nguyen TL, Inverso O, Fischer B, La Torre S & Parlato G, "Lazy sequentialization for TSO and PSO via shared memory abstractions", *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design. FMCAD Inc*, (2016), pp.193-200.

[12] Blom S, Darabi S, Huisman M & Oortwijn W, "The VerCors tool set: verification of parallel and concurrent software", *International Conference on Integrated Formal Methods*, (2017), pp.102-110.

[13] Eslamimehr M, Lesani M & Edwards G, "Efficient Detection and Validation of Atomicity Violations in Concurrent Programs", *Journal of Systems and Software*, (2017).

[14] e Silva RAB, Arai NN, Burgareli LA, de Oliveira JMP & Pinto JS, "Formal verification with frama-C: A case study in the space software domain", *IEEE Transactions on Reliability*, Vol.65, No.3, (2016), pp.1163-1179.

[15] Mandrykin M & Khoroshilov A, "Towards deductive verification of concurrent Linux kernel code with Jessie", *Computer Science and Information Technologies (CSIT)*, (2015), pp.5-10.

[16] Zhang X, "Debugging Multithreaded Programs Using Symbolic Analysis", Software Testing, Verification and Validation (ICST). (2017), pp.557-558.

[17] Shi Q, Huang J, Chen Z & Xu B, "Verifying Synchronization for Atomicity Violation Fixing", I*EEE Transactions on Software Engineering*, Vol.42, No.3, (2016), pp.280-296.

[18] Uspenskiy S, "A survey and classification of software testing tools", *Master of Science Thesis, Lappeenranta University of Technology*, (2010).

[19] Boogerd C & Moonen L, "Prioritizing software inspection results using static profiling", *Source Code Analysis and Manipulation*, (2006), pp.149-160.

[20] Etenting Hub-Online Free Software Testing Tutorial. (n.d.), *Software Testing Verification*, 2018.