A Review of Formal Program Verification Tools Based on the Boogie Language

Inaam Ahmed, Theodore S. Norvell, and Ramachandran Venkatesan
Dept. Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland & Labrador
inaama@mun.ca, theo@mun.ca, venky@mun.ca

***Abstract*— Errors in the software are hazardous. Testing software, before its deployment, may catch some errors present in the software. However, testing does not guarantee their absence. Several formal verification tools have been built and used for formal verification to ensure the correctness of software in different programming languages. The Boogie language is a common intermediate representation for static verification of programs written in several high-level programming languages. In this paper, notable software verifiers such as Dafny, VCC, HAVOC, Verve, and Chalice are described and analyzed for similarities and dissimilarities with the HARPO verifier.**

***Index Terms*— Automated Verification, HARPO Verifier, Formal Verification**

I. Introduction

Software failure is fatal. It costs money and sometimes lives. The prevalent software design approaches are not mature enough to prove the correctness of software using rigorous testing. Even with rigorous testing, there are countless ways for a program to go wrong. While testing can be useful for finding errors in software, it can not be used to show their absence [1]. Therefore various forms of analysis can be used [2].

Sequential Programs: Sequential programs are hard to write correctly. The correctness of these programs primarily depends on implicit assumptions. For instance, a programmer can easily assume that a variable lies inside the array bounds, a function parameter is not null, contents of specific memory locations are not read by a method, data structures are accessed and manipulated with a specific protocol. How can a programmer reason for these assumptions? It is strenuous for them to verify that their assumptions are correct. Thus, writing correct sequential programs is tough.

Concurrent Programs: Concurrent programs are even harder to write. Data races, interference among threads, deadlocks and live-locks, data hiding and abstraction, and modularity are some of the challenges concurrent program writing incurs. The programmer must assume the effects of interference of a thread with all other threads currently running. Memory access is another challenge for a programmer; while one thread is accessing a memory location, other threads must not access the location concurrently. Locking mechanism enforce assumptions on locations protected by locks such as acquiring locks will not result deadlock.

The formal software verification approach can increase the productivity of a programmer and decrease the cost of dependable software production by reducing the cost of changing the software in and after development cycle. A standard approach for formal program verification is to use the automated theorem-proving technique [3].

Developing a verification tool is a process involved with complexity and criticality. It encompasses compilation technology, formal semantics, generation of verification conditions, translating the results into decisions and an interactive user interface. Source code together with program specifications in a higher programming language is translated into VCs. However, generating VCs for theorem provers is a complicated task. A common approach to deal with this complexity is to use an intermediate verification language. The verification process into two main steps: Translating the program specifications into an intermediate verification language (IVL) Boogie. Later, the Boogie source is converted into VCs and checked by the Boogie verifier to generate an error report.

Dafny, VCC, Chalice, Verve, and HAVOC are some of the tools that use Boogie and an intermediate verification language [3].

In this paper, an eventful backend of a formal verification system called Boogie Intermediate Verification Language is discussed, and number of efficacious verifiers based on Boogie backend are reported as follows: Section II describes the Hoare-style automated verifier named Boogie and Boogie Verification Language. Later sections describe the verifiers based on Boogie. Section III provides details of Dafny verifier; Section IV describes the concurrent C verifier named VCC; Section V describes a verifier named Chalice, and idea of permissions transfer approach for verification; Section VI describes a distinct verifier targeting only system software verification system named Verve; Section VII describes HAVOC, a scalable verifier for heap data structures. Section VIII describes Spec# verifier; Section IX describe AutoProof approach Eiffel using AutoProof; Section X describe an Extended Static Verifier which used same staging approach for verification conditions generation like rest of the verifiers in this paper used; Section XI describes our own verifier for HARPO language named HARPO Verifier with description of its design and evaluation. Section XII presents Conclusion.

II. BOOGIE

Boogie was developed by Hoare-Logic based program verifier[3]. Boogie language is an intermediate verification language used as a common intermediate representation in the verification of other higher-level programming languages. Boogie can also be used as an input/output format for the abstract representation and predicate abstraction. Internally, Boogie performs a series of transformations of source program into verification conditions to error report. Boogie, previously known as BoogiePL, as a language has imperative and mathematical components. A challenge is to create verification conditions Boogie verifier uses SMT solvers [4], such as *Z3*[1], to determine the truth of verification conditions.

III. DAFNY

The Dafny programming language is designed to write programs using built-in specification constructs. Dafny's treatment of locations is based on dynamic frame theory [5], [6]. Its compiler can produce executables for the .NET platform. Dafny attempts to determine the correctness of programs by checking the parts of the program for their own correctness and then infer the correctness of complete program based on smaller parts. The use of dynamic frames enables Dafny to prove the correctness when data-abstraction is used [8].

The Dafny programming language consists of imperative and sequential constructs, generic classes, dynamic allocation, and specification constructs. Dafny's specification constructs are pre-condition, post-condition, read set, write set, loop invariants and loop termination metrics. Types include algebraic, sets, and sequences. Ghost constructs and specifications and ignored when executable code is being generated. Dafny's verifier generates the Boogie and the Boogie tool is used to generate the verification conditions which *Z3* checks.[7]

The program verification process of Dafny is like many automatic verifiers using Boogie as their back end. A program written in Dafny is translated into Boogie and such that the correctness of the Boogie representation implies that the Dafny programs are correct [14]. Failures of VCs to verify are passed back to Dafny and processed to report meaningful errors and warnings in Dafny source code. Dafny was designed with verification in mind. And due to that, programs in Dafny are cleaner than programs in other verifiers such as VCC. Dafny is now a common choice for teaching and learning automated program verification.

IV. VCC

VCC (Verifier for Concurrent C) is layered on top of the C language for verification purposes. VCC was developed in Microsoft Hypervisor Verification Project (MHVP). The project was intended to provide verification of functional correctness properties of various software types, including commercial, system software, off-the-shelf software, and Microsoft Hyper-V. [8]

[1]Z3 us a state-of-art first-order theorem prover the support Satisfiability Modulo Theories (SMT).

VCC verification methodology is based on inline annotations in source code. Annotations are also called contracts for verification. These annotations are eliminated while generating the output of regular C compilation. However, for verification purposes, the output from preprocessor accompanying annotation supplied to VCC. VCC takes input and converts annotated C program into an internal representation for type checking and name resolution. Later, the internal representation is transformed i.e. simplifying the source, adding proof obligation, etc. The final output from transformation generates Boogie code. The VCC allows to add or remove transformations needed. The Boogie code generated from transformation does not contain information on imperative control flow, procedural and functional abstractions of transformed code. It contains large prelude which axiomatizes of C memory, ownership of objects, states of types, and arithmetic expressions, this part named prelude. The resultant source code is given to Boogie program verifier, which converts the program into set of verification conditions. These verification conditions are then passed to a theorem prover, in this case *Z3*, to be proved.

V. CHALICE

Chalice is an experimental language developed to allow the verification of concurrent programs. Language support various features, including dynamic objects and threads creation, locks, monitors, mutual exclusion, pre-conditions, and post-conditions. The language supports locking memory locations at very low-level using permissions. Chalice carries permissions and transfer of permissions approach to address the specification for verification of concurrent programs; permission models also allow Chalice to be used for stepwise refinement of program specifications. Refinement of pseudo-codes into more concrete implementation all becomes source code for Chalice verifier to check the correctness of refinement and specification. The programmer can specifically convey assumptions about the code explicitly via annotations. [9]

The verification methodology of Chalice centers around permissions and transfer of permissions. Chalice maintains a record of permissions for each memory location, and threads can access the memory location if they have sufficient permission. The framing problem is addressed with the permissions transfer approach. Threads must have enough permission to access a location. For instance in a thread, if caller demands callee to update location it has access on, the callee can not do unless it acquires the permission of that memory location from the same thread or thread must have sufficient permission to in order to call the callee. Permission is calculated as percentages; full permission is 100% corresponds to write-permission whereas read-permission is any percentage less than or equal to 100% and higher than 0%. The 'ϵ' is any positive fractional value between 0% and 100% represents the permission value [10]. Chalice programs are translated to Boogie and verification conditions are generated by Boogie verifier to be checked with *Z3* for their correctness.

VI. VERVE

High-level computer applications are created on top of low-level, such as operating systems and run-time language systems. The security and reliability of such lower-level software and system are critical. Errors can lead to system software crashes, data loss, and insecure hardware control. Verve is another formal verification system that uses Typed Assembly Language (TAL) with Hoare Logic to accomplish a highly automated verification. Verve primarily verifies the absence of many categories of errors in low-level code. The safety of assembly language instructions, run-time system, and every component of OS except boot loader can be verified with the Verve. Type and memory safety are verified with Verve, it has *"Nucleus"* which has access to hardware components such as memory. The *"Nucleus"* is implementation of memory allocation, garbage collection, interrupts and their handling, devices access control, and stacks. The kernel of OS later builds above the "Nucleus" and applications run on top of kernel. [11]

The verification methodology of Verve is breaking down the underlying system software and the run-time language system into two layers. The kernel application is written in C and compiled to TAL and checked with an already available TAL checker [12]. Nucleus mentioned above is written in assembly language, directly incorporating annotations. These annotations are better known as assertions such as loop invariants, pre- conditions, and post-conditions. These specifications in assembly language are translated into Boogie. Boogie verifier relies on Z3 that verifies the conditions automatically. In fact, one executable statement is translated into 2-3 lines of proof annotation.

VII. HAVOC

HAVOC (Heap-Aware Verifier Of C) is a static verifier for C similar to ESC/Java, and Spec#, static verifiers for Java and C#, respectively. HAVOC is a distinguished verifier deals with lower-level details of C language. It provides the automatic update for reachability predicate which is explicitly designed to deal with pointers and their arithmetic operations. HAVOC addresses on of the sources of unscalability for automatic verifiers struggling with the imprecision caused by a heap-allocated data structures. There are two fundamental correctness properties, control flow, and memory safety, which depend on the assertions pointing the contents of the heap data structure. These reachability predicates are required for specifying the properties of heap. [13]

HAVOC is designed for verification of C programs with the specification incorporated as annotation language in C. HAVOC interpret the annotated program into annotated BoogiePL [2] program. Boogie verifier generates the verification conditions and passes these on to Z3 for checking the truth of the verification conditions.

[2]BoogiePL is previous version of the Boogie.

VIII. SPEC#

Spec# is an object-oriented language created by extending C# with specifications features. Spec#'s goal is to provide more cost-effective ways to produce high-quality software. Notable annotations of Spec# are pre-conditions, post-conditions, object invariance, and non-null types. These annotations allow the programmer to write specifications expressing the intention of programmer about data and methods being used. Spec# compiler performs run-time checks to assert the specifications. [14]

Spec# performs both static and run-time checking. All specification annotations are labeled to differentiate the Spec# annotated and unannotated code. For checking the object invariance each class added with a new method that declares an invariant. Spec# underlying static checking is performed using Boogie. Boogie is responsible for checking specifications of a non-contract code, for example, it must determine that the non-contract code of a procedure meets its post-condition. BoogiePL code is simple implementation of basic blocks of statements from Spec# are assignments, assume, assert, and method call. These properties are added to the program for making assertions and assumptions on program statements. Boogie program goes into several translations and ends up with verification conditions. These verification conditions are checking with a theorem prover.

IX. EIFFEL AUTOPROOF

Programmers not comfortable with formal verification techniques experience difficulty while verifying the correctness of programs formally. Eiffel reduces the burden of writing enormous annotation specifications for program using Auto Proof approach. AutoProof is a static verifier for Eiffel programs and part of Eiffel Verification Environment (EVE) which encompasses numerous verification tool and utilize their synergy. [15]

Eiffel's verification with AutoProof is an automated translation of Effile program into Boogie program. Eiffel programs contain annotated contracts such as class invariance, pre-conditions and post-conditions. Some syntactical advances in Eiffel eliminate the need for many frame conditions. AutoProof generates standard annotations for automated verification to decrease the burden on the programmer. AutoProof relies on Boogie the same way that Chalice and Dafny does.

X. ESC/JAVA

Extended Static Checker for Java (ESC/Java) is a static compile-time program checker that catches errors in Java programs. ECS catches more errors than any conventional verifiers also a reason to be named extended. ESC can preferably catch error issues like null dereference, array out of bounds, type conversions. It also provides warnings on the synchronization issues like race conditions and deadlocks. Full functional program verification catches ideally all errors and their absence in a program at extreme cost whereas the static checking is less expensive but catches limited class

of errors. ESC lies in between two extremes of program verification domain. [16]

ESC is a modular checker perform checking by verification condition generation and automatic theorem proving. Each module is subroutine or a provided piece of code. Annotations are used to provide the specification of the routine being checked. ESC front end parses the java program with annotations and produces abstract syntax tree and type-specific background predicate for each class. The next stage translates each routine into guarded command, where each sub-command enforces that the parent command is true. The process of generating verification conditions has sharpened in ESC. Guarded commands in intermediate representation are not primarily represented in Boogie. However, they plays same role as Boogie. These commands are converted into verification conditions for checking.

XI. HARPO Verifier

HARPO (HARdware Parallel Object) project started in 2006 [17]. The mission of the HARPO project is to develop an industrially viable concurrent language that supports the wide variety of reconfigurable processor architectures and GPUs with functional correctness properties using automated verification [17], [18], [19]. This section contains information on HARPO Verifier's methodology and development. Some of the similar software verification tools are discussed from Section III to X are compared with HARPO Verifier in this section.

A. HARPO's Verification Methodology

The verification methodology of the HARPO verifier is significantly related to the previously mentioned verifiers. Since HARPO langauge is a concurrent language that is based of conditional critical sections and *rendezvous* between threads. It employs explicit transfer of permissions in order to verify concurrent programs. HARPO Verifier uses two different layers of operations while performing the verification of HARPO programs. Specifications are written in form of annotations in standard HARPO syntax. These annotations are ignored when compiling to C, VHDL, and CUDA. The HARPO Verifier transforms the annotated program into verification conditions by translating the program into Boogie. Later, Boogie Verifier transform the code into verification conditions and check for their correctness using SMT solver named *Z3*.

B. Annotation of Language

HARPO language is incorporated with annotation for its verification [20]. These annotations are assertions for reasoning the correctness of the program. Annotations contain pre-conditions, post-conditions, class invariants, loop invariants, claim specifications, assertions, and assumptions [21], [22], [23].

C. Design and Evaluation

HARPO verifier design consists of a parser, checker, code generator, error report processor. The parser creates the AST with valid HARPO program syntax. The checker performs name resolution, type checking, and creation. The code generator takes AST and generates the equivalent code in Boogie [7,20]. The Boogie verifier takes Boogie code and transforms it into verification conditions. The verification condition is checked with Z3. Later, the verification errors are processed by an error processor to refer back the error in HARPO source code. Testing the functionality of the HARPO Verifier is performed with unit tests (commands and expressions), system testing is performed for some examples [24].

HARPO verifier used Dafny's inline annotations methodology. The idea of permissions transfer is tested in Chalice, and HARPO Verifier is using that idea of permission transfer differently for verification concurrent programs. All verifiers reported in this paper employe Boogie and Z3 as their underlying verification methodology.

XII. Conclusion

In this paper, some automated verifiers designed with on staging approach for generating verification conditions are reported and provide a motivation for HARPO Verifier. Each verifier targets different areas of software verification problems. However, all of them use the same underlying verifier for generating verification conditions and checking them with SMTs. Boogie is known and proved to be a beneficial resource for verification condition generation and checking. Dafny uses dynamic frame theory and targets the verification problems involving data abstraction and modular specifications. VCC is layered on top of the C language and is used for verification of C programs with inline annotations of specifications and targets the verification of commercial and system software. The Chalice was an experimental language and used the idea of permission and transfer of permission for verification; Chalice featured the automated verification of concurrent programs using transfer of permissions. Verve is a distinguished verifier targeted with lower-level system software verification using two layers approach, Nucleus, and kernel. Verification of programs containing pointers is a challenge for programmers, and HAVOC is an excellent resource for verifying programs heap data allocation; it reduces the chances of imprecision due to dangling pointer references. Spec# is easily adoptable verification system as it is extension to an already in-demand language C#; programmers can easily accept the change and take advantage of specifications and verification for software solutions. Eiffel reduced the efforts of making explicit annotations for dynamic frames using AutoProof. ESC/Java is not exactly using the Boogie language but similar staging as rest of the static verifiers mentioned. ESC/Java project targeted the widely accepted language for object-oriented software development, and it is much closer to static checking with the capability of addressing numerous error categories. HARPO Verifier is addition to list of static verifiers targeting the verification of concurrent high-level programming language designed to target reconfigurable, GPUs, and microprocessors.

ACKNOWLEDEMENTS

This research was supported in part by funds from the National Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] E. Dijkstra, "The humble programmer", Communications of the ACM, vol. 15, no. 10, pp. 859-866, 1972.

[2] O. Hasan and S. Tahar. (2015). Formal Verification Methods. In M. Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology, Third Edition (pp. 7162-7170). Hershey, PA: IGI Global. doi:10.4018/978-1-4666-5888-2.ch705

[3] K.R.M. Leino, This is Boogie 2, Microsoft Research, Tech. Rep., 2008, draft. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=147643

[4] M. Leonardo and B. Nikolaj "Z3: An Effecient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems." Ed. By C. R. Ramakrishnan and R. Jakob. Vol. 4963. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin, Apr. 2008. Chap. 24, pp. 337-340. isbn: 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

[5] I. T. Kassios. "The dynamic frames theory. In: Formal Aspects of Computing" 23 (3 2011), pp. 267-288. issn: 0934-5043. doi: 10 . 1007 /s00165-010-0152-5.

[6] W. Benjamin "Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction". PhD thesis. Karlsruhe Institute of Technology, 2011.

[7] K.R.M. Leino and V. Wstholz, (2014). "The Dafny integrated development environment." arXiv preprint arXiv:1404.6602.

[8] Vertisoft XT: The Verisoft XT project. http://www.verisoftxt.de (2007)

[9] K.R.M. Leino, P. Mller, and J. Smans, Verification of concurrent programs with Chalice, in Foundations of Security Analysis and Design V, ser. LNCS, vol. 5705, 2009.

[10] B. John, "Checking interference with fractional permissions. In Radhia Cousot, editor, Static Analysis" 10th International Symposium, SAS 2003, volume 2694 of Lecture Notes in Computer Science, pages 5572. Springer, June 2003.

[11] Y. Jean and C. Hawblitzel. "Safe to the last instruction: automated verification of a type-safe operating system." ACM Sigplan Notices 45.6 (2010): 99-110.

[12] J. Chen, C. Hawblitzel, F. Perry, M. Emmi et al."Type-preserving compilation for large-scale optimizing object-oriented compilers." SIGPLAN Not., 43(6):183192, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375604.

[13] S. Chatterjee, S.K. Lahiri, S. Qadeer, et al. (2007) "A Reachability Predicate for Analyzing Low-Level Software." In: O. Grumberg, M. Huth (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2007. Lecture Notes in Computer Science, vol 4424. Springer, Berlin, Heidelberg

[14] B. Mike, K. Rustan M. Leino, and S. Wolfram. "The Spec programming system: An overview. In Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)" volume 3362 of Lecture Notes in Computer Science, pages 4960. Springer, 2004.

[15] J. Tschannen, C.A. Furia, M. Nordio, et al. (2011). "Verifying Eiffel programs with Boogie." arXiv preprint arXiv:1106.4700.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge et al. Extended static checking for Java. In PLDI, pages 234245. ACM, 2002.

[17] T.S. Norvell, "Language design for CGRA project. design 8." [unpublished draft], Memorial University of Newfoundland, 2013.

[18] T.S. Norvell, A.T. Md.Ashraful, L.Xiangwen, Z. Dianyong, HARPO/L: A language for hardware/software codesign. in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2008.

[19] T. S. Norvell, A grainless semantics for the HARPO/L language in Canadian Electrical and Computer Engineering Conference, 2009.

[20] T. S. Norvell, "Annotations for Verification of HARPOL. Draft Version 0." [unpublished draft], Memorial University of Newfoundland 2014.

[21] I. Ahmed, T.S. Norvell, R. Venkatesan, "Verifying the correctness of HARPO Programs in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2018.

[22] Y.G. Fatemeh, "Verification of the HARPO language Masters thesis, Memorial University, 2014.

[23] T. S. Norvell, "HARPO/L: "Concurrent Software Verification with Explicit Transfer of Permission" in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2017.

[24] I. Ahmed, T.S. Norvell, R. Venkatesan, "Design and Verification of Counter Using HARPO Programming Language" in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2019.