Design and Verification of a Counter using HARPO Programming 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*—HARPO (HARdware Parallel Objects) is a concurrent programming language designed to run on coarse-grained reconfigurable computing architectures, Field Programmable Gate Arrays (FPGAs), and Graphical Processing Units (GPUs). The HARPO verifier translates programs into Boogie for verification. Writing specifications and annotations in the HARPO language for a hardware design shortens the development time and bridges the gap between high-level programming languages and hardware description languages. In this paper, the design of an integer *"Counter"* is verified using HARPO Verifier by translating the program into Boogie. Various *counting* scenarios containing explicit transfer of permissions have been verified using HARPO verifier.**

HARPO Verifier, FPGA, Counter, VHDL

I. Introduction

HARPO project aims to target variety of coarse-grained reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs); Graphical Processing Units (GPU), and modern microprocessors [1]–[3].

Testing is not sufficient to ensure the correctness of programs and this motivates us to find methods of program analysis verification [7], [8]. The design of HARPO allows programs to be annotated for verification [4]. The HARPO Verifier has been developed to verify the correctness of sequential and concurrent HARPO programs using the Boogie verifier [5] under the covers. Concurrent programs are verified using the explicit transfer of permissions methodology [6].

In this paper, the design and verification of a simple hardware component, namely an integer *counter* is reported. A generic *counter* class is written in HARPO programming language using annotations for its verification. Fig 1 shows the data flow diagram of the HARPO Verifier. The HARPO Verifier shares the same front-end with other HARPO backends [9]. The parsing phase creates an abstract syntax tree (AST) from the source code. All syntactical errors are reported to an Error Recorder. Later the checker phase performs name resolution, type creation, and type-casting. Again all semantic errors are reported to Error Recorder. An updated AST from checking phase goes to code generator which performs translation of HARPO code into Boogie [9], [10]. The code generator traverses the AST and generates Boogie code. The Boogie code serves as an intermediate representation (IR) of the HARPO code for generating Verification Conditions (VCs). Boogie verifier generates VC from standard axioms and functions embedded in customized IR. Boogie verifier generates uses the *Z3* SMT solver [11] to attempt to prove the VCs. The Error Report processor takes all verification errors and feeds them to Error Recorder after mapping verification errors to their corresponding HARPO source code.

Acknowledgement: The research was supported in parts by funds from National Science and Engineering Research Council (NSERC) of Canada.

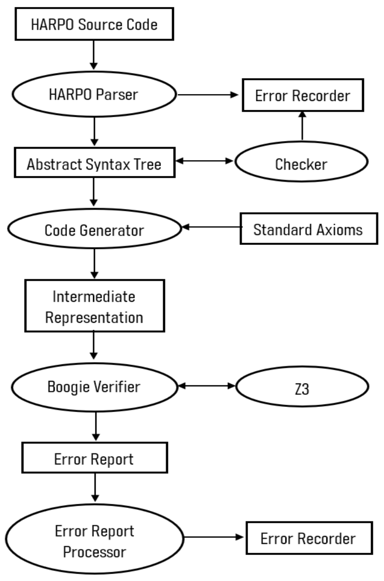

Fig. 1. Data Flow of HARPO Program Verification

The remainder of this paper is organized as follows: Section II describes some of the notable verifiers using Boogie language as their intermediate representation. Section III describes the *counter* design problem. Section IV describes the abstract syntax trees of *counter* class. Section V describes checking phase of verification. Section VI describes code generator. Section VII briefly explains actions of the Boogie verifier. Section VIII explains the error report processor. Section IX has conclusions with suggested future work.

II. Related Work

Numerous static verifiers have been developed in the past few decades [12]–[22]. These verifiers primarily use verification conditions generation at the heart of static verification process. Boogie is a verifier [5] for the Boogie Intermediate Verification Language (IVL) that is used for intermediate representation of several static checking verifiers. The Dafny programming language was developed for verification in mind and being used as high-level programming language for developing verified programs and generate .NET executables [13]. Chalice is one of the first static verifiers used the idea of permissions to verify concurrent programs [14]. VCC was developed as a verification layer on top of C for verifying concurrent C programs [16]. Verve is a complete operating system verified by using Boogie as verification conditions generator [17]. HAVOC is heap-aware verifier designed to efficiently verify the correctness of C programs using heap data structure with less effort [19]. C# language programs are verified using Spec# as an easily adoptable technology [20]. Eiffle uses AutoProof verifier to address frame problem by auto-generating frame conditions [20]. ESC/Java is a static checking using same staging strategy that other verifiers are using; however ESC/Java does not use Boogie besides has its own verification conditions generator [22].

III. "Counter" Design

A HARPO program in Listing 1 has a class named *Counter*. An integer type variable named *count* has been initialized to *'0'* and it can be assigned with in a range i.e. {*-2147483648,...,+2147483647*}. Generally, the purpose of this counting scenario is to increment the value of the *count* variable every time an event occurs.

The *increment* procedure is implemented by thread **t0** to update the *count* field. In HARPO, each thread and object has, at each point in time, a certain amount of permission on each location. The total permission at any time on one location can not exceed 1.0. A thread needs 1.0 permission on an object in order to write to it. Initially each *Counter* object claims *0.5* permission on the *count* field and the class invariant asserts *count* as readable[1]. The *increment* procedure takes the *0.5* permission from the calling thread. The assignment of *count* to *count+1* is inside a *with* command. The *with* command locks the object, and allows the thread to use the locked object's permission as defined in class invariant. Thus, the assignment has an error; the total permission that the thread can use can be shown to be greater than 0.5, but can not be shown to be 1.0. Thread **t1** invokes *increment* procedure and responsible to explicitly transfer the permission mentioned in *takes* clause of *increment* procedure. In this case, **t1** also does not claim any permission and unable to provide required permission to **t0**. So this is a second error which the verifier should detect and report.

[1]Readable and writable are two different levels of access to a particular location, these access levels are presented with amount of permissions they possess on that location. For instance, full permission i.e. *1.0* only allows writability and any amount of permission between greater than *0.0* and less than *1.0* is readability.

```
(class Counter()
  obj count: Int32 := 0
  claim count@0.5
  invariant canRead(count) /\ count >_ 0
  proc increment()
   takes count@0.5
   pre count>_0
   post count'>0
   gives count@0.5
 (thread (*t0*)
   (while (true) do
       (accept increment()
         (with this do
            count := count+1;
          with)
        accept)
     while)
  thread)
 (thread (*t1*)
    increment();
  thread)
class)
```

Listing 1. *Counter* Class in HARPO Programming Language

IV. Abstract Syntax Trees

The first phases of translation produce an abstract syntax tree (AST) for the program to represent the essential inform ation of the program. A simplified AST is shown in Listing 2.

```
[ClassDeclNd(
 [ClaimNd[ClaimNd#0](
  PermissionMapNd[PermMapNd#1](
   [ LocSetNd( NameExpNd( count ) : loc{Int32} ) ],
   [ FloatLiteralExpNd( 0.5 ) : Real64 ] ) ),
  ClassInvNd[*inv*0](
   BinaryOpExpNd( AndOp,
    CanReadOp( LocSetNd( NameExpNd( count ) : loc{Int32} ) ): loc{Int32},
    ChainExpNd([ GreaterOrEqualOp ],
     [ FetchExpNd( NameExpNd( count ) : loc{Int32} ) : Int32,
       IntLiteralExpNd( 0 ) : Int32 ] ): Bool )
  MethodDeclNd(PublicAccess)
     [ PreCndNd(
        ChainExpNd(
           [ GreaterOrEqualOp ],
           [ FetchExpNd( NameExpNd( count ) : loc{Int32} ) : Int32,
             IntLiteralExpNd( 0 ) : Int32 ] )
           : Bool ) ]
     [ PostCndNd( /*Conditional Expression*/ ) ]
     [ GivesPerNd(
        PermissionMapNd[PermMapNd#3](
           [ LocSetNd( NameExpNd( count ) : loc{Int32} ) ],
           [ FloatLiteralExpNd( 0.5 ) : Real64 ] ) ) ]
     [ TakesPerNd( /*Permission Map*/ ) ]
  ObjDeclNd[count](Ghost :false,
     NamedTypeNd( Int32 ) : loc{Int32},
     ValueInitExpNd( IntLiteralExpNd( 0 ) : Int32 ) : Int32 ),
  ThreadDeclNd[t#0](WhileCmdNd( BooleanLiteralExpNd( true ) : Bool,
   AcceptCmdNd([ MethodImplementationDeclNd( increment,
    WithCmdNd( ThisObjRef( this ) : NONE,
     AssignmentCmdNd( [ NameExpNd( count ) : loc{Int32} ],
      [ BinaryOpExpNd( AddOp,
        FetchExpNd( NameExpNd( count ) : loc{Int32} ) : Int32,
         IntLiteralExpNd( 1 ) : Int32 ) : Int32 ]
   ) ) ) ]) ) ) ] ),]
```

Listing 2. A Simplified AST of *Counter* Class in Listing 1

V. Checker

The checker phase of verification takes AST generated from the parser and edit the AST considering the semantics of HARPO language such as, linking names to their declaration, creating and checking types, and inserting type conversions. Listing 2 is completely checked AST and ready to be used in the next phase. Each expression is annotated with its final result type; blue words in Listing 2 are types created during the checking phase. For instance, the assignment command in the *Counter* class uses a value that is of *Int32* type.

VI. Code Generator

Standard axioms and function are embedded in the start of Boogie code. These axioms define the values integers and reals can take, and types need to be used in customized code. The framing problem is addressed at the code generation phase [23], [24]. The code generator takes the AST and converts the declaration into constants; it declares a constant for each field defined in the HARPO code; it converts each thread to Boogie procedures. The output generated by code generation is not intended to faithfully encode the execution of HARPO programs; this would be impossible, as BOOGIE is a sequential language which HARPO is concurrent; rather, our goal is to ensure that the generated Boogie code will verify if and only if the HARPO code is error free. Boogie language provides *assert* statement to encode the proof obligations from source language and *assume* statement to guarantee the properties provided by the source language such as values of minimum and maximum values of *permission* variable. While generating the code each line of Boogie code is given line number, and error messages on specific guarded statements are set. These line numbers are error messages that help identify the location of errors in source file. For assignment statement in Listing 1 intermediate representation will generate the assertion:

```
129: assert LockPermission[This_Counter,Counter.
     count] + Permission[This_Counter,Counter.count
     ] == 1.0;
```

Listing 3. Guard Statement Taken From IR of Listing 1

The permission of the thread and the *Counter* object are summed and checked for equality to 1 in order to check whether the thread has access to sufficient permission to write the location. It can be inferred that the thread's own permission is 0.5. However all that can be inferred about the permission belonging to the locked object is that it is greater than 0. Thus the total permission is not known to be 1.0 and this assert command will result in an error. If the object invariant were changed to state that the object's permission is 0.5, this assertion would verify.

VII. Boogie verifier

The Boogie verifier is a static verifier intended to generate verification conditions and check them using *Z3*. *Z3* checks the correctness of the VCs and reports back the results in an error report. For Listing 1, HARPO verifier, gets the error report from Boogie verifier in Listing 4. The Boogie verifier says the assertion on *line:129* might not hold. This assertion given in Listing 3 asserts the total amount of locked object permission, and the permission of current thread is running on is equal to 1.

```
input(129,25): Error BP5001: This assertion
     might not hold.
Execution trace:
    input(99,5): anon0
    input(99,5): anon4_LoopHead
    input(102,9): anon4_LoopBody
    input(108,13): anon5_Then
    input(113,21): anon3
Boogie program verifier finished with 1
     verified, 1 error
```

Listing 4. Output From Boogie verifier for Listing 1's IR

VIII. Error Processing

Error report generated by the Boogie verifier is processing using Error Processor. Error Processor parses the output string and gets the line number of errors and their messages. We separate the errors into three different categories in Error Recorder, such as verification errors, fatal errors, and warnings. All errors and their traces form Boogie verifier are inserted as verification errors in the Error Recorder. However, they are always preprocessed before inserting them into Error Recorder.

The line numbers of the Boogie code are mapped to line numbers in HARPO code. The lines containing assertion commands are linked to mapped to error messages that will make sense to the HARPO programmer. For instance error captured in listing 4 is marked on *line:129* and error trace shows the root of error from *line:113* back to *line:99*. *Line:129* has *assert* statement and associated message with line number corresponding to HARPO code. Error Processor select line number and verification error is feed into Error Recorder. Therefore, after verification Error Recorder contains verification error and its line number.

IX. Conclusion and Future Work

The verification of the *Counter* class shown above illustrates some of the capabilities of the HARPO verifier to find errors in concurrent code. We have tested the verifier on a number of other examples, as well as carefully unit testing each part of the system.

Future work will include automatically inferring certain loop invariants so as to lessen the burden on the programmer in annotating their code.

References

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

[2] 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.

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

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

[5] 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

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

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

[8] 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

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

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

[11] 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.

[12] I. Ahmed, T.S. Norvell, R. Venkatesan, A Review of Formal Program Verification Tools based on Booogie Language in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2019.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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

[20] 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.

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

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

[23] 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.

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