Design and Verification of a Counter using HARPO Programming Language

Inaam Ahmed
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland & Labrador
inaama@mun.ca

T.S. Norvell and R. Venkatesan
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland & Labrador
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 Graphicaal Processing Units (GPUs). The HARPO compiler translates programs into Boogie for verification and into VHDL for implementation on FPGA. Writing specifications and annotations in 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. The correctness of design must be verified before translating and implementing the design on FPGA.**

***Index Terms*— HARPO Verifier, FPGA, Counter, VHDL**

I. INTRODUCTION

HARPO project started in 2006 aimed to target variety of coarse-grained reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs); Graphical Processing Units (GPU), and modern microprocessors [1]–[3]. HARPO language design is part of HARPO project, build with program specifications and verification in consideration [4]. HARPO Verifier is developed to verify the correctness of HARPO sequential and concurrent program using static verifier named Boogie [5]. Concurrent programs are verified using explicit permissions transfer methodology [6]. Testing is not sufficient to reason the correctness of program and this motivates the formal program verification [7], [8]. Static verification of HARPO programs is aimed to check the correctness without executing the program.

In this paper, the design and verification of a simple hardware component, namely 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. HARPO Verifier shares the same front-end with other HARPO backends [9]. HARPO parsing phase creates an Abstract Syntax Tree (AST) from the source code. All syntactical errors are reported to 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 that performs translation of HARPO code into Boogie [9], [10]. The code generator traverses the AST inside out and generates the Boogie code. The generated Boogie code is also called 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 the error report by interacting with SMT solver *Z3* [11]. 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.

This paper is organized as follows: Section I contains an introduction to HARPO project. Section II contains some of the notable verifiers using Boogie language as their intermediary representation for verification purposes. Section III describes the *counter* design problem from scratch. Section IV describes the abstract syntax trees of designed *counter* class. Section V describes checking phase of verification. Later, Section VI describes code generator. Section VII briefly explains Boogie's verification conditions generation. Section VIII also briefly explains the Boogie verifier. Section IX contains error report processor. Section X 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. If program contains errors then static checking must report some errors, called soundness of checking, and every reported error must be genuine error called completeness of static checking. Boogie is Hoare-style verifier [5] provide Boogie Intermediate Verification Language (IVL) that is used for intermediate representation of several static checking verifiers. 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 farmer static verifiers used the idea of

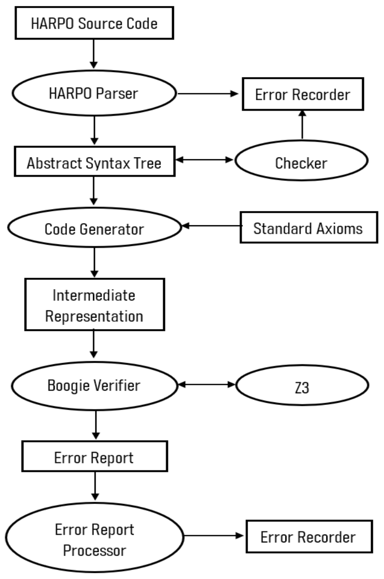

Fig. 1. Data Flow of HARPO Program Verification

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. Initially, *counter* class claims *0.5* permission on the *count* field and class invariant assert *count* as readable[1]. The *increment* procedure takes the *0.5* permission from callee and set the pre-condition of increment to a positive *count* value. The assignment of *count* to *count+1* is inside lock implemented with current object being referenced using a*with* command. The *with* command uses locked object's permission defined in class invariant to update the count field. The thread has no claim for permission on *count* field, and the only permission it received with the locked object *0.5* which is not sufficient to perform the assignment. Thus, this increment must fail, and our verifier must report the reason of this failed attempt to write *count* variable. The *accept* command is exclusively implementing the *rendezvous*. It can accept multiple implementations, but goes forward non-deterministically with one implementation from the list. The target of this problem verification process is to get detailed error report from Boogie verifier in order to improve the design specifications. 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**.

[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

AST of any program in HARPO compiler and verification is a syntactic structure of HARPO programs used for generating intermediate verification code. 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

Checker phase in verification takes AST generated from the parser and edit the AST considering the semantics of HARPO language such as, linking names to their declaration, create checking types, find locations, and type conversion. Listing 2 is completely checked AST and ready to be used in the next phase. The expression is annotated with their final result type, blue words in Listing 2 are types and expressions created while checking phase. For instance, the assignment command in *Counter* class has used resultant value that must be of *Int32* type. Similarly, boolean expression needs to have the *Bool* type resultant value. All types colored *"blue"* are assigned by checker based on the semantics of HARPO language.

VI. CODE GENERATOR

Standard axioms and function are embedded in the start of Boogie code. These axioms sets rules of code being generated, such as limits of 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]. Code generator takes the AST and converts the declaration into constants, find all names in AST and make fields by mapping HARPO types to Boogie types. Convert all threads to Boogie procedures, and implementations accepted by thread are converted into *goto* statement in Boogie. The output being generated by code generated rather than implementation-specific, it is more specification oriented for generating VC. 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 point of error in source language. For assignment statement in Listing 1 intermediate representation will generate the assertion as :

```
129: assert LockPermission[This_lock ,Counter.count]
     + Permission[This_Counter ,Counter.count] == 1
     .0;
```

Listing 3

GUARD STATEMENT TAKEN FROM IR OF LISTING 1

The **t0** thread permission of *counter* class and assumed permission from class invariant are summed up and checked for the total permission to equal to *"1"* for making this assignment happen. Since the total amount of permission thread has and the locked object is not equal to *"1"*, this guarded statement generates proof not satisfying the condition of assignment, and hence this example must fail to make assignment. If **t1** explicitly transfer the required permission to **t0** then the generated code will pass the verification.

VII. VCS GENERATION

VCs generation involves various design decisions. Boogie architecture provides a separation between source language encoding with its implementations. Boogie programs are completely sequential. The conversion of concurrent programs into Boogie also involves various design level decisions. The idea to convert the source language program into Boogie is to make the process of VC generation uncomplicated and an automated. The idea of translating the HARPO code into Boogie has simplified the process of VC generation; because Boogie code is converted into VC by Boogie verifier. In short, the input if Boogie verifier is kind of program instead of formulas and that program is converted into logical formulas. VC generation with executable program statements also depends on the declarations of the source program. The incorporation of background predicate i.e. *standard axioms* in Boogie program helps on generating the proof for particular statements in source language program. For instance *count* field in Listing 1 is declared with *Int32* type and *Int32* has the range of {*-2147483648,...,2147483647*}. The properties of HARPO are no longer used by the Boogie instead it uses the properties defined in the intermediate representation.

VIII. BOOGIE VERIFIER

Boogie verifier is a Hoare-style static verifier intended to generate the verification conditions and check them using *Z3*. It checks the correctness of the VCs and reports back the results in and error report. Boogie Verifier checks the error conditions defined by the specifications in source programs such as pre-conditions, post-conditions, casting errors and other programming methodologies. For Listing 1, HARPO verifier, get the error report from Boogie verifier in Listing 4. The Boogie verifier says the assertion on *line:129* might not hold.

```
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

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"*.

The reason for failure to pass this assertion, locked object and the current object on which thread is running are the same. The total amount of permission after summation is *0.5*. The caller of *increment* procedure must provide *0.5* permission to pass this assertion. The amount of permission shown in 1 is encoded in one of our standard axioms in Boogie as 2. These permission values are simple fractions. Listing 5 shows standard axiom encoded in Boogie for Eq 1.

$$\forall \;\; p \in R,\, 0 \;\leq p \leq \, 1, \;\; where \;\; R \;\; is \;\; real \tag{1}$$

```
type Perm = real;
const unique minPerm : Perm;
axiom minPer == 0.0;
const unique maxPerm : Perm;
axiom maxPer == 1.0;
function isValidPermission(Perm) returns (bool);
axiom (forall x: Perm:: isValidPermission(x) <==>
     minPerm <= x && x <= maxPerm);
```

Listing 5
An Axiom For Valid Permission Value

IX. Errors 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 fed as Verification Error in Error Recorder. However, they are always preprocessed before feeding them into Error Recorder.

A. Errors and Warnings Mapping

The line numbers of the Boogie code point to the line numbers in HARPO code. The lines containing guarded statements are linked to their error messages. 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.

X. Conclusion and Future Work

The automated translation of *Counter* class to Boogie programs is tested for each declaration using unit tests. We have also verified various examples using different *counter* class design specifications. In the future, we plan to: uuse error messages for correcting the code, verify three different control transfer scenarios exploiting the synchronization mechanisms among the threads, introduce permission type in HARPO and permission values will be extended to various degrees. HARPO Verifier can verify program's implementing concurrent access to memory where concurrent assignment to memory locations is critical.

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

[8] [2 Bib:FVMethods] 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] [22 Bib:FatemehThesis] 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] [21] 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] [8 Bib:verisoftVCC] 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.