

Verification of Concurrent HARPO Programs

Using Formal Verification Theory

by

© *Inaam-Ahmed*

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of *Engineering*

Department of *Electrical and Computer Engineering*
Memorial University of Newfoundland

October 2020

St. John's

Newfoundland

Abstract

Defects in software cost money and sometimes lives. Even with rigorous testing, there are countless ways for programs to go wrong. Testing does not guarantee that a given program is correct for every input. Concurrent program testing does not guarantee that a program is correct for the example inputs. Formal program verification has been used as a technique to ensure program correctness for several years. It analyses the properties of the code and ensures that nothing goes wrong. In this thesis, a formal verification tool is designed and implemented based on Boogie IVL (Intermediate Verification Language) for a multi-threaded and object-oriented language named HARPO (HARdware Parallel Objects). We have designed the specific annotations intended only for use in verification. These annotations help the HARPO verifier to translate the program into an equivalent Boogie code. The resulting Boogie code is checked with theorem provers to verify the correctness of the HARPO program. Boogie code generation is tested using unit testing, and some of the case studies are reported. Consequently, programmers can use the HARPO verifier for verification of their concurrent programs.

Acknowledgements

“I would like to thank my professors, Dr. Theodore Norvell and Dr. R. Venkatesan, for their intellectual and practical assistance, advice, encouragement, and monetary support. I would like to say thanks to former graduate student Fatemeh Yousefi Ghalehjoogh for developing the formal verification theory.

I dedicate this thesis to my beloved parents.”

— Inaam Ahmed

Contents

Abstract	ii
Acknowledgements	iii
List of Tables	x
List of Figures	xi
0 Introduction	2
0.1 Problem Description	2
0.2 Motivation For Research	3
0.3 Research Objective	4
0.4 Contribution	5
0.5 Thesis Outline	6
1 Software Verification Background	7
1.1 Program Verification	7
1.1.1 Formal Program Verification	8
1.1.2 Theorem Proving	9

1.1.3	Symbolic Simulation	9
1.1.4	Model Checking	10
1.2	Hoare Logic	10
1.3	Separation logic	11
1.4	Dynamic Frames Theory	12
1.4.1	Implicit Dynamic Frames	14
1.5	Verification of Concurrent Programs	15
1.6	Boogie Based Formal Verification Tools	16
1.6.1	Dafny	16
1.6.2	Chalice	17
1.6.3	VCC	18
1.6.4	HAVOC	18
2	HARPO Language Design	20
2.1	HARPO Programming Language	20
2.2	HARPO Language Design	21
2.2.1	Program	21
2.2.2	Types	22
2.2.3	Variables	24
2.2.4	Classes and Interfaces	25
2.2.4.1	Class Members	26
2.2.5	Statements	27
2.2.6	Literals	31
2.2.7	Implicit Conversion	32

2.3	Annotations for Verification of HARPO	33
2.3.1	Permission Model	34
2.3.2	Classes and Interfaces	35
2.3.3	Threads	36
2.3.4	Statements	37
2.3.5	Operations	39
2.4	Permission Maps	39
3	Boogie Language	41
3.1	Overview	41
3.2	Declarations	42
3.3	Boogie Name Space	43
3.4	Built-in Types	44
3.5	User-Defined Types	44
3.5.1	Type Constructor	45
3.5.2	Type Synonyms	45
3.6	Axioms	46
3.7	Constants and Functions	47
3.8	Expressions	48
3.9	Procedures	50
3.10	Assertions and Assumptions	51
3.11	Havoc	52
3.12	Control Transfer	52
3.13	Conditional Blocks and Loops	53

4 HARPO to Boogie Translation	54
4.1 Boogie Prelude	55
4.1.1 Class and Interface Type	55
4.1.2 Global Variables	56
4.1.3 Axioms and Functions	57
4.2 Customized Translation	61
4.2.1 Class and Interface Translation	62
4.2.2 Fields and Constants Translation	62
4.2.3 Class Invariant Translation	63
4.2.4 Class Claim Translation	64
4.2.5 Class Constructor Translation	65
4.2.6 Thread Translation	66
4.2.7 Method Translation	67
4.2.8 Expression Translation	69
4.2.9 Command Translation	71
4.2.9.1 Skip Command Translation	71
4.2.9.2 Sequence Command Translation	71
4.2.9.3 Local Declaration Command	72
4.2.9.4 Assignment Command	73
4.2.9.5 If Command	74
4.2.9.6 While Command	76
4.2.9.7 For Command	77
4.2.9.8 Assert and Assume Command	80
4.2.9.9 Call Command	82

4.2.9.10	With Command	85
4.2.9.11	Co Command Parallelism	88
5	HARPO Verifier Design	91
5.1	Compiler Overview	91
5.2	HARPO Verifier Overview	93
5.2.1	Front-end Compilation	94
5.2.2	Abstract Syntax Tree Components.	94
5.2.3	Checker Passes	95
5.2.4	Boogie back-end	96
6	Automated Verification	99
6.1	HARPO Source Code	100
6.2	Translation	101
6.2.1	Parser	101
6.2.2	Checker	104
6.2.3	Code Generator	106
6.3	Boogie Source Code	107
6.4	Boogie Error Report	111
6.5	Code Generator Tests	113
6.5.1	Class Declaration	113
6.5.2	Interface Declaration	113
6.5.3	Object Declaration	114
6.5.4	Ghost Object Declaration	114
6.5.5	Constant Declaration	115

6.5.6	Ghost Object Initialization	115
7	Conclusion and Suggested Future Work	116
7.1	Conclusion	116
7.2	Future Work	117
	Appendices	123
A	Automated Translation Case Studies	124
A.1	Code Generation	124
A.1.1	Type Conversion	124
A.1.2	Initialization with Boolean Expression	125
A.1.3	Chained Initialization Expression	126
A.1.4	Initialization Expressions	126
A.1.5	Class Claim Specification	129
A.1.6	Assert Command	129
A.1.7	Local Variable Initialization	130
A.1.8	Bool Type Local Variable	131
A.1.9	Assume Command	131
A.1.10	If Command	132
A.1.11	While Command	133
A.1.12	For Command	134
A.1.13	Call Command	135
A.2	Counter	137
A.2.1	Counter Version 0	139

A.2.2	Counter Version 1: without class invariant	144
A.2.3	Counter Version 2: without locking mechanism	149
A.3	Buffer: reading and writing	154
A.4	Permission Transfer Scenario 1	163
A.5	Permission Transfer Scenario 2	177
B	Tables and Figures	194

List of Tables

2.1	HARPO Primitive Types Widening Relationships	23
2.2	HARPO Operators and their Top Down Precedence	31
4.1	User Defined Functions for Checking Defineness	70
4.2	Expression Translation	70
B.1	Primitive Types Translation Map	194
B.2	HARPO Language Reserved Words	195
B.3	Boogie Reserved Words	195
B.4	Operators Types Translation Map	196

List of Figures

2.1	Overview of HARPO compiler where dotted arrows indicate the final outputs.	21
2.2	WP(Write Permission) split into multiple RP (Read Permissions)	34
5.1	HARPO Compiler Implementation Overview	92
5.2	Translation Data Flow	93
5.3	Parsing character stream into token stream and generating the AST	95
5.4	Attributed AST	96
5.5	Top level UML Diagram containing parent and child nodes inside AST Package	97
5.6	Checker Phase	98
5.7	Complete Boogie Backend	98
6.1	Verification Data Flow	99
6.2	Abstract Syntax Tree of <i>Counter</i> Class	102
A.1	Counter Class Diagram	138
A.2	Counter Class Message Sequence Diagram	138
A.3	Buffer Class Diagram	155

A.4	Buffer Class Message Sequence Diagram	156
A.5	Scenario 1 Class Diagram	163
A.6	Scenario 1 Message Sequence Diagram	164
A.7	Scenario 2 Class Diagram	177
A.8	Scenario 2 Message Sequence Diagram	178
B.1	Complete Package Diagram of HARPO Compiler	197

Chapter 0

Introduction

0.1 Problem Description

Software failure can be fatal. We entrust even our lives to information and complex algorithms where safety and security are critical. The increasing complexity of software products with demand of shortened development cycles and higher customer expectations have placed a major responsibility on software testing and verification. The prevalent software design approaches are still immature, thus the area of software verification is demanding more work to write verified software. Software plays an essential role in safety-critical and mission-critical systems. Following are some historical failures where software was blamed:

1. The erroneous angle-of-attack sensor reading in Ethiopian Airline Boeing 737 Max 8 triggered the anti-stall software system causing the plane to hit the ground at 575 miles per hour in 2019. Software requirements were partially blamed for 157 deaths reported in the accident. The software was not designed to behave

correctly in the presence of sensor failure [1].

2. The loss of the 286 million US dollar was incurred on Hitomi satellite which spun around due to altitude control failure [2].
3. Badly-written Toyota software resulted in unintended acceleration problem for several years is blamed for 89 deaths and several recalls.[3]
4. US Airline lost 65 million US dollars when airline reservation system reported flights as sold out [4].
5. Four patients died, and two were left with life-long injuries due to erroneous software controlling the Therac-25 radiation machine. It was entirely relying on the computer for its safe operation [5].
6. In 1999, an Arian rocket self exploded after 40 seconds of its launch due to implementation errors present in the program of inertial guidance system [6].
7. Denver Airport opening was delayed, partly due to the malfunctioning baggage sorting system and missed its fourth opening deadline [7].

Historical software failures and unexpected behaviors encourage programmers and research and development teams to find techniques to determine the correctness of software before deployment.

0.2 Motivation For Research

Testing a program with any sets of inputs can reveal the presence of errors, but not their absence. Thus testing is insufficient to ensure systems are error-free[8]. An

automated software verification system helps developers detect faults in a program, which may lead to runtime errors. The purpose of program verification is to provide an error-free software system. A critical system needs to be known to be defect-free with mathematical certainty prior to being deployed. Formal program verification techniques consider computer programs as mathematical objects, and their properties are verified by mathematical proofs [8].

The static 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 the development cycle. 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. [9, 10, 11, 12]

0.3 Research Objective

HARPO (HARdware Parallel Objects) is an object-oriented, concurrent programming language. It is a behavioral programming language for writing a wide variety of hardware design configurations. HARPO’s object-oriented paradigm supports concurrent and parallel computing features. HARPO is intended to target a variety of hardware platforms, including microprocessors, FPGA (Field Programmable Gate Array), and CGRA (Coarse-Grained Reconfigurable Architecture). A HARPO source program may be compiled to the equivalent C or VHDL (VHSIC Hardware Description Language)[10].

In this thesis, the design of the HARPO verifier and its implementation are re-

ported. The verifier is designed based on the verification methodology developed by F. Y. Ghalehjoogh, which is reported in [12]. The HARPO verifier translates HARPO program specifications into equivalent Boogie programs. Its underlaying verification methodology is based on the Boogie. The Boogie verification tool takes the Boogie source program and uses a theorem prover SMT¹ solver *Z3* to detect the errors in source code [13]. This verification methodology for program verification based on theorem provers guarantees the correctness of programs hundred percent.

0.4 Contribution

Numerous verification tools have been designed to verify sequential and concurrent programs in the past few years based on Boogie, including *Dafny*, *Chalice*, *VCC (Verifier for Concurrent C)*, *Eiffel*, and *Spec#* [14, 15, 16, 17, 18]. The HARPO verifier is also designed on top of Boogie. We use an approach similar to that used in the high-level verifiers mentioned above. Following are notable contributions:

- HARPO syntax has been updated to support annotation for verification.
- Verified some concurrent HARPO programs using the HARPO verifier based on techniques presented in this thesis.
- Verification errors reported by the Boogie verification tool are translated back to HARPO errors pointing to the corresponding source code in HARPO.

¹SMT (Satisfiable Module Theories) refers to the problem of determining that a first-order formula is satisfiable.

- The idea of permissions and transfer of permissions have been tested in implementation of HARPO verifier.
- HARPO verifier described in this thesis determines that the *safety* property holds for tested HARPO programs.

0.5 Thesis Outline

Chapter 0 describes the statement of the problem, motivation for this research, and our contribution. Chapter 1 describes the formal program verification background. Chapter 2 provides the HARPO language background and annotations used for verification of HARPO. Chapter 3 describes the Boogie language syntax and semantics primarily used in our translation. Chapter 4 describes in detail the translation of HARPO to Boogie with examples. Chapter 5 describes the detailed design of the HARPO verifier. Chapter 6 has automated translation examples and unit testing of the code generator. Chapter 7 concludes with future work. Some case studies are reported in Appendix A and other some useful tables and figures provided in Appendix B.

Chapter 1

Software Verification Background

1.1 Program Verification

Since program testing is insufficient to prove program correctness, whereas program verification can ensure correctness to produce error-free programs. The design process of the software contains verification to check whether or not the software will behave as desired. In Dijkstra's words,

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." [8]

The verification of large programs with traditional paper and pencil method to find the correctness of even a small sequential program is difficult and error-prone. To overcome this complexity, computer-based software tools were introduced. The purpose of program verification is to provide error-free software.

1.1.1 Formal Program Verification

Since testing is insufficient to give proof of error-free system, formal verification methods are applied to prove the absence of errors. These methods use foundations of theoretical computer science such as automata theory and strongly typed logic calculi to deal with complex programs. By using formal program verification, the correctness of programs is guaranteed relative to abstractions that are assumed. Notable formal verification methods include theorem proving, symbolic verification, and model checking. In this thesis, we have adopted only the theorem-proving technique. [19]

Formal program verification considers computer programs as mathematical objects, and their properties are verified by mathematical proofs. Ultimately, formal methods and mathematical techniques are used to verify software-specifications to provide error-free software. Program verification can also be performed with the help of model checking or symbolic verification. However, the mathematical method of the program verification proves the correctness of programs hundred percent by using theorem provers according to supplied verification conditions. The theorem provers consist of standard properties such as axioms and primitive-inference rules. Programming languages *C++* or *Java* are hard to be used in theorem proving where we require a language with logical primitives and syntax such that programs in the intended language are described using unambiguously defined rules and semantics. [14, 20]

1.1.2 Theorem Proving

Theorem proving, also referred to as automated reasoning is a widely accepted formal verification technique. Programs are converted into logical formulas, and properties of these formulas are verified using computer-based tools by converting the program into underlaying verification conditions called theorem provers. Formal logic has made theorem proving a very flexible verification technique.

Proof of logical formulas may require human interaction or manual proof efforts, and its complexity depends upon underlying logic. Propositional logic, first-order logic, and higher-order logic are examples of these underlying logic variants. Among these, higher-order logic is the most demonstrative form of logic. It allows quantification over sets and functions and needs user input to verify formulas expressed in higher-order logic. Based on human intervention, theorem proving has been divided into two broad sub-branches: Automated Theorem Proving and Interactive Theorem Proving. [21]

1.1.3 Symbolic Simulation

Traditional testing approaches are performed with constant values. Symbolic verification bridges the gap between traditional verification techniques and advanced formal verification. In this type of verification, symbols are used in the simulation. For example, consider the problem of verifying an eight input AND gate. Instead of verifying it with 256 different test vectors, the boolean variable is applied over each input simultaneously, and output is recorded and checked to see if it is equal to $var0 \wedge var1 \wedge var2 \wedge var3 \dots \wedge var7$. [22]

1.1.4 Model Checking

Model checking is used to verify types of systems that are reactive in nature and depend on time and the environment. The idea of model checking is to build a model representing the states of the given system and then verify the given property of each modeled state. A finite state model is developed and tested against the given specifications. The application of model checking is limited to systems expressible as a finite state machine. [23]

1.2 Hoare Logic

C. A. R. Hoare developed a deductive reasoning methodology used to guarantee the correctness of programs [8]. Hoare logic is based on deductive reasoning and provides a set of axioms and rules to reason about the properties of programs. Programs developed with their design requirements are required to fulfill the expected behavior. Program requirements are explicitly specified by assertions on the values of variables at the start and end of execution. A Hoare triple is a program P , its pre-condition (R) and its post-condition (E) as given below:

$$R \quad \{P\} \quad E , \tag{1.1}$$

which is partial correctness where R is an assertion on the initial state, and E is an assertion on the final state, which is true after the execution of the program. Program termination is not considered in Hoare's triples. The assignment rule is the fundamental rule of Hoare Logic as it was developed for imperative programming languages. An inference rule is developed in Hoare logic using the following assignment

substitution:

$$R[e/x] \quad \{x := e\} \quad R , \quad (1.2)$$

where R is post-condition with $R[e/x]$ pre-condition formula after substitution of e for all occurrences of x .

The framing problem deals with how we can determine which program properties are unchanged during an assignment. It arises during verification with Hoare logic. Specification of the assignment tells which kind of change needs to be performed on certain variables, whereas the rest of the program variables are unchanged. Considering the example in which m is not changed by the program:

$$(m == 100) \quad \{n := 200\} \quad (n == 200 \wedge m == 100) , \quad (1.3)$$

where the problem of scalability arises when reasoning about all the unchanged program variables for a program part. Another problem comes while reasoning about heap containing program variable(s) modified in one program section and another variable that is not modified in the same program. Framing problem is one of the inherent challenges in which parts of heap remain unchanged during some operations. A few strategies, namely, separation logic, dynamic frames, and implicit dynamic frames, are used to resolve the framing problem. [24]

1.3 Separation logic

Separation logic is an extension to Hoare logic. It essentially introduced a new logical operation to indicate the disjointness of heaps and index aliasing. As the name implies, the idea behind separation logic is to perform spatial conjunction¹ such that

¹Spatial conjunction denoted with ‘*’ in Equation 1.4 shows the disjoint parts of heap.

its sub-formulas hold for disjoint portions of the heap. Hoare logic is improved with separation logic by adding new inference rules, such as shown in eq. 1.4.

$$\frac{(\{R\} \quad C \quad \{E\})}{(\{R * Q\} \quad C \quad \{E * Q\})} \quad (1.4)$$

As given in eq. 1.4, we can reason about C locally and then extend specifications with variables and locations which are not modified by C at all. This way, the frame problem can be addressed with less effort. [25]

1.4 Dynamic Frames Theory

Kassios developed dynamic frames theory to address the frame problem in object-oriented programs in which modularity and encapsulation make framing complex [26, 27]. Consider a problem where a program \mathbf{P} is $y := \text{true}$; and a strong post-condition is the case of program having another variable z will be:

$$y == \text{true} \wedge z == \text{old}(y) , \quad (1.5)$$

however it is not possible to reason such a strong post-condition because we do not know z or old values of y . In modular programs, change in one module does not make any change in another module. So, modular specifications can be written separately and can be verified separately as well.

Encapsulation hides data implementation from clients. It is another property of the object-oriented approach, which makes specifications more complicated. With multiple Holder objects it is hard to reason about each object which encounters frame problem. For example, Listing 1.4.1 is an example program in HARPO hiding data

field *a*.

```
(class Holder()
  obj a : Int32 := 0;
  proc get(out x : Int32)
  proc set(in y : Int32)
  (thread t0
    (accept
      get(out x:Int32)
      (x := a;)
      |
      set(in y:Int32)
      (a := y;)
      accept)
    thread)
  class)
```

Listing 1.4.1: Data Hiding Example in HARPO

Footprints are the locations accessed by a method or function. The dynamic frame theory applied to object-oriented programs abstracts the footprint of statements. Footprints are applied by pure functions, which are called dynamic frames. *Holder* class can be modified with the following pure function:

```
(class Holder()
  obj a : Int32 := 0;
  proc footprint(out &a)
  proc get(out x : Int32)
    pre canRead(footprint())
  proc set(in y : Int32)
    pre canWrite(footprint())
    post get(x) = temp;
  (thread t0
    (accept
      get(x) then (return x;)
      |
      set(y) then (x := y)
      |
      footprint() (return &a)
      accept)
    thread)
```

```
class )
```

Listing 1.4.2: Application of Footprints using Pure Function.

1.4.1 Implicit Dynamic Frames

Dynamic frame methodology requires the frame annotations to be written for each method. Implicit dynamic frames use a specification style with access permissions and eliminate the need for explicit writing of frame annotations. Here frame can be inferred from the specifications. Access assertions are used to represent permission to access a location [28]. Listing 1.4.3 shows implicit dynamic frame implementation in HARPO.

```
( class Holder()
  pre acc(x) /\ x = 0;
  private obj x : Int32 = 0;
  proc set (in temp : Int32)
    pre acc(x)
    post acc(x) /\ temp == x;
  (thread t0
    (accept set(int temp : Int32)
      (x := temp)
      accept)
  thread)
class )
```

Listing 1.4.3: Implicit Dynamic Frame

The *set* method require modification access to *x* so *acc(x)* takes permission from caller and give it to callee. However, the constructor does not need access permission because, upon the creation of a new object, the permission is provided by the system to the constructor.

1.5 Verification of Concurrent Programs

In concurrent computing, time-sharing is used to allocate multiple tasks on a processor. Concurrent programming refers to when a single program executes on multiple processors using multi-threading. Concurrent programming increases the performance and flexibility of the program, but reasoning about the correctness of a concurrent program(s) becomes more complex. Characteristics of multi-threaded programs make it even harder to write correctly. Scheduling of threads is not deterministic, and reasoning is complicated due to the interference of other threads.

Correctness properties of the concurrent programs are either safety properties, liveness properties, or a conjunction of both kinds. Safety properties state that nothing wrong will happen during the execution of a thread. Liveness properties state something good will eventually happen. For instance, a safety property named mutual exclusion ensures that only one process can execute the critical section at one time. Mutual exclusion is usually implemented with locks [29].

An implicit dynamic frame theory can handle data races but only supports full access permission, i.e., Full-Permission or No-Permission. Extending implicit dynamic frames and fractional permission enable concurrent programs verifications in *Chalice*. The characteristic of a reading permission is that it will avoid the writes but not reads, whereas write permission is full permission and prevents all other writes and reads. The *Chalice* language verifier uses the same technique. For reading a particular memory location, any non-zero fraction is required, whereas write requires full permission. Thus fractional permission can be transferred and divided, but no two threads can have the same write permission on the same location at once. Fractional

permission can be combined with the implicit dynamic frame, as shown in the listing below.

```
class Container()
{
    var x, result : int;
    void method Add()
        requires acc(result,1) /\ acc(x,0.5)
        ensures acc(result,1) /\ acc(x,0.5)
    {
        result := result + x;
    }
}
```

Listing 1.5.1: Reading and Writing Permissions

1.6 Boogie Based Formal Verification Tools

A program verification tool ensures the developers that their code satisfies its specifications for every possible input and thread scheduling. Numerous verification tools have been designed over the past few years based on Boogie's verification methodology, including *Dafny*, *VCC*, and *HAVOC*. Boogie generates first-order verification conditions for given specifications. These verification conditions are given onward to SMT solver for checking the validity of these conditions [13]. The Boogie verification debugger does provide verification error information in detail that can be used to get the right source of error in a program.

1.6.1 Dafny

The *Dafny* verifier fulfills the need for formal verification using a programming style named *design-by-contract*. It is easy to learn and incorporates many features of useful programming language, which are based on the imperative programming paradigm.

During a program verification process, states and properties are checked at the entry and exit of a method in a program. The *Dafny* verifier uses Boogie in the back-end, whereas the annotations in *Dafny* are written in a format that is close to a high-level programming language. The *Dafny* verifier converts specifications and code into an intermediate verification language named Boogie-IVL (Boogie Intermediate Verification Language).

Dafny's verification provides user-defined functions and ghost variables to verify programs. Usually, modules are verified separately, and the *Dafny* verifier only knows about the interfaces of modules visible to their caller modules [30]. Since the back-end of *Dafny* is dependent on the Boogie IVL, the results of the Boogie imply the results of verification in *Dafny*. *Dafny* has become a general-purpose tool for verification of programs.

1.6.2 Chalice

Chalice is a language and program verifier having its verification methodology based on permission and their transfer when required. Concurrent programs are harder to verify due to data races, deadlocks, and interference among threads. In *Chalice* programmers explicitly introduce annotations inside code. *Chalice* detects bugs present in the concurrent program by analyzing the annotated program and verifying that the given annotations are not violated. [16].

1.6.3 VCC

The Verifier for Concurrent C (*VCC*) is used for verification of concurrent C programs. Programs are written in the C language with annotations incorporated as pre-conditions, post-conditions, and invariants based on a design-by-contract approach. Contracts specify the impact of functions they are written for. The caller of the function must validate the contract of the function necessary to make that call. Programs written in the C language translated into the Boogie IVL and the Boogie code is used to generate verification conditions to determine the correctness of the C program.

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 incorporates annotation supplied to *VCC*. *VCC* takes input and converts the annotated C program into an internal representation for type checking and name resolution. Later, the internal representation is transformed by simplifying the source, adding proof obligation, etc. The final output from transformation generates Boogie code. The resultant source code is given to the Boogie program verifier, which converts the program into a set of verification conditions. These verification conditions are then passed to a theorem prover, in this case, *Z3*, to be proved [17]

1.6.4 HAVOC

HAVOC(Heap-Aware Verifier Of C) is a verifier used to determine the correctness of programs written in C; it targets system software verification. It is designed to be a

modular verifier, which is the reason *HAVOC* is a good foundation for building robust safety checks. *HAVOC* is designed for verification of C programs with the specification incorporated as annotations in C. *HAVOC* interprets the annotated program into an annotated Boogie program. The Boogie verifier generates verification conditions and passes to Z3 for checking the validity. [18]

Chapter 2

HARPO Language Design

2.1 HARPO Programming Language

HARPO is an object-oriented, concurrent programming language developed in 2008 [10]. It is a high-level behavioral programming language for writing a wide variety of hardware design configurations. HARPO's object-oriented paradigm supports concurrent and parallel computing features. A HARPO program targets a variety of hardware platforms, including microprocessors, FPGAs, CGRAs, and GPUs. The design of the HARPO compiler is shown in Figure 2.1. The HARPO source program is compiled to C and VHDL.

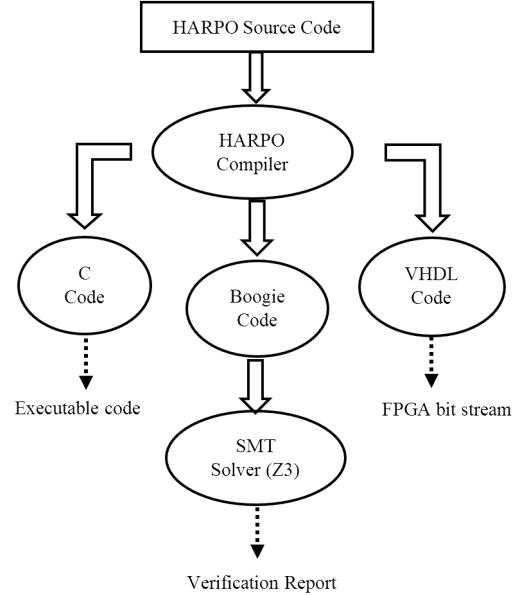


Figure 2.1: Overview of HARPO compiler where dotted arrows indicate the final outputs.

2.2 HARPO Language Design

This sub-section contains HARPO language design reported in the latest language design version-10 [9]. Table B.2 in appendix contains all HARPO language keywords¹.

2.2.1 Program

A HARPO program consists of a sequence of classes, interfaces, constants, and objects.

$$\text{Program} \longrightarrow (\textit{ClassDecl} \mid \textit{IntfDecl} \mid \textit{ObjectDecl} \mid \textit{ConstDecl} \mid ;)^* \textit{eof}$$

¹For differentiation in text, all HARPO keywords are italicized, and Boogie Keywords are bold-faced whereas in production rules both HARPO and Boogie keywords are boldfaced.

2.2.2 Types

The HARPO data model contains locations, objects and arrays. Locations hold values of the primitive types. Objects have fields that are locations, arrays, or references to objects. Arrays have items that are locations, arrays, or references to objects.

- Primitive Types: are types that represent the sets of values. Primitive types do not contain mutator to change the values. However, changing the values is possible with an assignment to objects of primitive types. HARPO primitive types are:

- **Integer:** *Int8, Int16, Int32, Int64*
- **Real:** *Real16, Real32, Real64*
- **Boolean:** *Bool*

Type	Range
<i>Int8</i>	$\{-128, \dots, 127\}$
<i>Int16</i>	$\{-32\text{.}768, \dots, 32\text{.}767\}$
<i>Int32</i>	$\{-2\text{.}147\text{.}483\text{.}648, \dots, 2\text{.}147\text{.}483\text{.}647\}$
<i>Int64</i>	$\{-9\text{.}223\text{.}372\text{.}036\text{.}854\text{.}775\text{.}808, \dots, 9\text{.}223\text{.}372\text{.}036\text{.}854\text{.}775\text{.}807\}$
<i>Real16</i>	$\{-6.10e + 5, \dots, 6.55e + 4\}$
<i>Real32</i>	$\{-3.4e + 38, \dots, +3.4e + 38\}$
<i>Real64</i>	$\{-1.7e + 308, \dots, +1.7e + 308\}$
<i>Bool</i>	$\{true, false\}$

Declaration of an integer type field having name *temp*. Types have the widening relationship as given in Table 2.1.

```
obj temp : Int32 := 0;
```

	<i>Int8</i>	<i>Int16</i>	<i>Int32</i>	<i>Int64</i>	<i>Real16</i>	<i>Real32</i>	<i>Real64</i>
<i>Int8</i>		✓	✓	✓	✓	✓	✓
<i>Int16</i>			✓	✓	✓	✓	✓
<i>Int32</i>				✓	✓	✓	✓
<i>Int64</i>					✓	✓	✓
<i>Real16</i>						✓	✓
<i>Real32</i>							✓
<i>Real64</i>							

Table 2.1: HARPO Primitive Types Widening Relationships

- Classes: Classes contain methods that may change the object state. Following is a simple *Math* class containing *count* field and thread **t0**:

```
(class Math()
  obj count: Int16 := 0
  proc increment()
    pre count >= 0
    post count' = count + 1
  (thread (*t0*)
    (accept increment
      count := count+1;
    accept)
  thread)
class)
```

Listing 2.2.1: Simple Math Class

- Interfaces: contain method declarations and fields but not threads.
- Arrays: contain locations, arrays or references to objects. Arrays have one dimension and indexed from 0 to the *Bound* expression where bound must be constant expression.

$$\text{Bound} \rightarrow \text{IntegerConstant}$$

- Generic Types: allow to write types that function differently based on their specific placeholder value. Generic types must be instantiated in order to be used.

2.2.3 Variables

Variables are named locations, arrays, or named references to objects. The type may not be generic, and a missing type is inferred from the initialization expression.

$$ObjectDecl \longrightarrow (\text{const} \mid \text{obj}) \ Name[: \ Type] := InitExp$$

Declaration with defined or inferred types creates a named location when type is primitive. For array types, declaration creates an array of locations.

$$\begin{aligned} InitExp &\longrightarrow Exp \mid ArrayInitExp \mid \text{new } Type(CArg^*) \mid (\text{if } Exp \text{ then } InitExp \mid \text{else } \\ &\quad \text{if } Exp \text{ then } InitExp \mid \text{else } InitExp [\text{if}]) \end{aligned}$$

$$ArrayInitExp \longrightarrow (\text{for } Name: Bounds \text{ do } InitExp [\text{for}])$$

$$CArg \longrightarrow Exp$$

- For an object of primitive type e.g. *Int32* or *Real64*, the initialization expression should be compile time constant having type assignable to the *Type*.
- Array objects are initialized with *ArrayInitExp*.
- Constructors accept objects or compile-time values as arguments.
- *Type* of *InitExp* must be sub-type of the *Type*

2.2.4 Classes and Interfaces

Classes are primary constructs of the HARPO language. Classes can be generic with one or more generic parameters.

$$\begin{aligned} ClassDecl \longrightarrow & (\text{ class } Name \text{ } GParams^? \text{ } (CParam}^+ \text{)} \text{ } (\text{ implements } Type^+)^? \\ & (ClassMember)^* [\text{class } [Name]] \) \end{aligned}$$

- *Name* represents name of class.
- *GParams* are used for the generic classes.
- *Type(s)* are interfaces implemented by the class.

An interface is a generic or non-generic type. One or more generic parameters are allowed for generic interfaces.

$$\begin{aligned} IntfDecl \longrightarrow & (\text{interface } Name \text{ } GParam}^? \text{ } (\text{ extends } Type}^+)^? (IntfMember)^* \\ & [\text{interface } [Name]]) \\ IntfMember \rightarrow & Field \mid Method \mid ; \end{aligned}$$

- *Name* is name of the interface.
- *GParams* are the generic parameters.
- *Type(s)* are the types interface extends.
- Constructor parameters represent connected objects. These parameters are connections with names to other objects. For example object *n* knows the object *m* by the name *x* as given below:

(class Math (**obj** $x : P$) ... **class**)

obj $m = \text{new } P()$

obj $n = \text{new } \text{Math}(m)$

2.2.4.1 Class Members

ClassMember \longrightarrow *Field* | *Method* | *Thread* | ;

- **Fields:** Fields are named locations, arrays, or references to objects within objects.

Field \longrightarrow *Access*? *ObjectDecl*

Access \longrightarrow **private** | **public**

- **Methods:** Methods can be declared anywhere in the class but implemented only inside the scope of a thread.²

Method \longrightarrow *Access* **proc** *Name*(*Direction* *Name* : *Type*)^{*})

Direction \rightarrow **in** | **out**

- **Threads:** Threads execute with the context of an object. One object may contain more than one thread; multiple threads provide the required concurrency.

Thread \rightarrow (**thread** *Block* [**thread**])

²HARPO does not have a declare-before-use rule. Everything is checked inside out for names during resolution.

2.2.5 Statements

A block is a sequence of statements.

$$Block \rightarrow Command\ Block \mid Local\ Declaration \mid ;\ Block \mid$$

- *Assignment Statements*

$$Command \rightarrow (ObjectIds) := (Expressions)$$

$$Command \rightarrow ObjectId := Expression$$

$$ObjectIds \rightarrow (\underline{ObjectId})^+,$$

$$Expressions \rightarrow (\underline{Expression})^+$$

$$ObjectId \rightarrow Name \mid ObjectId [Expression] \mid ObjectId.Name$$

- *Local Variable Declaration*

Local declarations are local in scope to the block that follows them. Local declarations can be constant that may not be assigned. Names of the declarations must be unique with in the same thread.

$$LocalDeclaration \rightarrow (\mathbf{obj} \mid \mathbf{const})\ Name[\underline{: Type}] := Exp\ Block$$

- *Method Call Statements*

A method call can be a call to local method or call to a method member of another object.

$$Command \rightarrow ObjectId.Name(Args) \mid Name(Args)$$

$$Args \rightarrow [\underline{Expressions}]$$

- Sequential Control Flow

Command → (**if** *Expression* **then**

$$\begin{aligned} & \text{Block } (\underline{\text{else if }} \text{Expression } \underline{\text{then }} \text{Block})^* (\underline{\text{else }} \text{Block})^? [\underline{\text{if}}] \\ & | (\underline{\text{while }} \text{Expression } \underline{\text{do }} \text{Block } [\underline{\text{while}}]) \\ & | (\underline{\text{for }} \text{Name} : \text{Bounds } \underline{\text{do }} \text{Block } [\underline{\text{for}}]) \end{aligned}$$

- Parallelism

For parallel blocks execution two vertical bars are used for differentiating parallel blocks where *Bounds* must be a constant.

$$\begin{aligned} \text{Command} \rightarrow & \left(\underline{\text{co }} \text{Block } (\underline{\parallel } \text{Block})^+ [\underline{\text{co}}] \right) \\ & | \left(\underline{\text{co }} \text{Name} : \text{Bounds } \underline{\text{do }} \text{Block } [\underline{\text{co}}] \right) \end{aligned}$$

- Method Implementation

Command → (**accept** *MethodImpl* (| *MethodImpl*)^{*} [**accept**])

MethodImpl → *Name*((Direction *Name* : *Type*)^{*},)

[*Guard*] *Block*₀ [**then** *Block*₁]

Guard → **when** *Expression*

Methods are declared outside the threads and implemented inside of a thread.

More than one method can be implemented in one thread enabling the *rendezvous*. When a thread reaches an *accept* command, it waits for a call from another thread before executing the method implemented inside the *accept*.

Some standard restrictions apply to the method implementation as follows:

- Types and directions of arguments must match the declared signature.
- Each declared method is implemented once per class, and no implementation is acceptable without the declarations.
- The *Guard* expression must return a boolean value.
- The *Guard* may not refer to any *out* parameters.

- *Locks*

Command —→ (**with** *Exp* [*Guard*] [**do**] *Block*[**with**])

Locks are implemented inside *with* command using object(s) implementing *lock* interface. The *Guard* must be a boolean expression and if not then by defaults *true*, and *Exp* is the object of type implementing locking interface. *Guard* and *Block* both execute atomically with respect to the other *with* sharing the same *lock*. When the *Guard* expression returns *false* the *lock* will be unlocked and everything starts again with the semantics of *lock*.

- Expressions

$$Exp \rightarrow Exp0 \underline{((=|<=|<=>) } \underline{Exp0}^* [\text{as } Id]$$

$$Exp0 \rightarrow Exp1 \underline{(\backslash / | or) } \underline{Exp1}^*$$

$$Exp1 \rightarrow Exp2 \underline{(/ \backslash | and) } \underline{Exp2}^*$$

$$Exp2 \rightarrow Exp3 | not Exp2 | \sim Exp2$$

$$Exp3 \rightarrow Exp4 \underline{((=| \sim =| not =| <| <-| >-| >) } \underline{Exp4}^*$$

$$Exp4 \rightarrow Exp5 \underline{((+| -) } \underline{Exp5}^*$$

$$Exp5 \rightarrow Exp6 \underline{((\ast| / | div | mod) } \underline{Exp6}^*$$

$$Exp6 \rightarrow -Exp6 | Exp7$$

$$Exp7 \rightarrow Exp8' | Exp8$$

$$Exp8 \rightarrow Primary$$

$$Primary \rightarrow (Exp) | ObjectId | Literal$$

$$| canRead(locSet) | canWrite(locSet) | permission(ObjectId)$$

$$| acc(PermMap) | length(ObjectId)$$

Expressions has operators precedence as top to down shown in Table 2.2. All expressions must not be evaluated by particular order e.g left to right or right to left, and, associativity of the associative operators may not be followed.

Chaining the operators made by the sequence of the operators *and* together.

For example and expression:

$$a \sim b < c = d$$

is logically equivalent to the expression:

Type	Operator(s)
Arithmetic	mod, div, /, *, -, +
Logical	and, /\ , or , \/
Relational	$=, \sim =, <, >, -<,>-$
Logical	$=>, <=, <=>$

Table 2.2: HARPO Operators and their Top Down Precedence

$$a < b \text{ and } b > c \text{ and } c \sim = d$$

2.2.6 Literals

Decimal integer literals are formed with series of decimal digits with required (any) number of internal underlines for readability e.g. “9”, “245”, ”1_000_000”. Integers with base 2,8, and 16 are allowed and written with their base first followed by ‘#’ symbol and the number that makes literal

$$\text{DecimalLiteral} \rightarrow D \mid D(\underline{D} \mid _)^*D$$

$$\text{HexLiteral} \rightarrow 16\#(\underline{H} \mid H(\underline{H} \mid _)^*H$$

$$\text{OctLiteral} \rightarrow 8\#(\underline{O} \mid O(\underline{O} \mid _)^*O$$

$$\text{BinLiteral} \rightarrow 2\#(\underline{B} \mid B(\underline{B} \mid _)^*B$$

$$D \rightarrow 0 \mid 1 \mid \dots \mid 9$$

$$H \rightarrow 0 \mid 1 \mid \dots \mid 9 \mid a \mid \dots \mid f \mid A \mid \dots \mid F$$

$$O \rightarrow 0 \mid 1 \mid \dots \mid 7$$

$$B \rightarrow 0 \mid 1$$

Floating point literals must be maximum sixty three bits long are defined with the following rule,

$$\begin{aligned}
 \text{RealLiteral} &\rightarrow . \text{ DecimalLiteral } [\underline{\text{Exp}}] \\
 &\mid \text{ DecimalLiteral } . [\underline{\text{Exp}}] \\
 &\mid \text{ DecimalLiteral } . \text{ DecimalLiteral } [\underline{\text{Exp}}] \\
 &\mid \text{ DecimalLiteral } \text{ Exp} \\
 \text{Exp} &\rightarrow (\underline{\text{e}} \mid \underline{\text{E}}) [+ \mid -] \text{ DecimalLiteral}
 \end{aligned}$$

2.2.7 Implicit Conversion

- Real

The type of any real literal can be of *Real16*, *Real32*, or, *Real64*. Initialization of such an object is defined when *InitExp* is completely a real literal, and the object has given type explicitly defined. Then there is an implicit ‘as’ performed. For example, the following two objects are equivalent.

```

obj p : Real16 := 0.938
obj q : Real16 := 0.938 as Real16

```

- Integer

All integer literals are by default *Int32* as long as value is under $2^{31} = 2,147,483,648$. For greater value it should be of *Int64* type. When an object is typed explicitly and literal is also of integer type then literal is typed as type of object. For example given initializations are same:

```
obj p : Int16 := 1
```

```
obj q : Int16 := 1 as Int16
```

The “*as*” construct forces the literal value to be reinterpreted as another type following it. However, it can only be applied if the expressions are of arithmetic type.

2.3 Annotations for Verification of HARPO

Annotations for verification are incorporated in the HARPO syntax [31]. These annotations are necessary for writing the specifications. However, the use of annotations must be meaningful for validating program specifications and generating valuable error reports. Ghost fields that are ignored by other backends such as C and VHDL and only used in verification.

As discussed in Chapter 0, a standard approach for program verification is to use the theorem-proving technique. Source code with program specifications in a high-level programming language is converted into the verification conditions. However, generating verification conditions for theorem provers is a complex task. A common approach to deal with this complexity is to use an intermediate verification language. For the HARPO verifier, we have mitigated the complexity by dividing the verification process into steps. First, we automatically translate the HARPO program into Boogie code, and second, use Boogie to generate the verification conditions.

2.3.1 Permission Model

The permission model of HARPO is reported in [32]. In this section the refined summary of permission model is described:

- Each thread and object holds a certain nonnegative amount of permission on each location at each point in time during execution.
- The sum of all these permissions on any given location never exceeds 1.0.
- The permission can be transferred between objects and threads and from thread to thread.
- To read a location a thread needs access to non-zero permission on it.
- To write a location, a thread needs access to permission of 1 on it.

One full permission can split into multiple read permissions show in Figure 2.2. Read permission can be unlimited as long as we can represent the smallest floating point number on underlaying machine architecture.

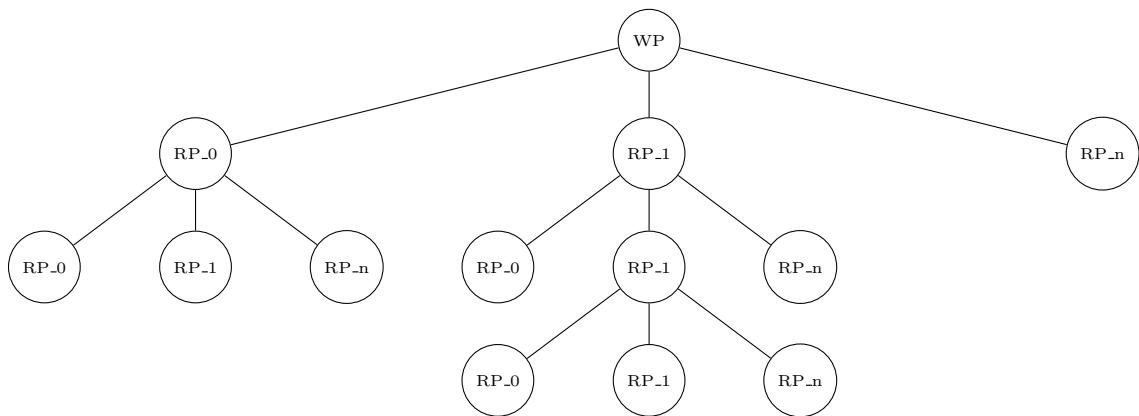


Figure 2.2: WP(Write Permission) split into multiple RP (Read Permissions)

2.3.2 Classes and Interfaces

HARPO programs may contain one or more classes; each class may have its *claim* and class *invariant* annotations. Instantiations of classes can be done several times to create objects or pass objects as parameters to other class constructors for building connections between objects. Objects are always initialized during their initialization, and reassignment is not allowed.

- Claim: acquire the initial ownership by object on its fields.

$$Claim \rightarrow \mathbf{claim} \ PermMap$$

- Invariant: must hold initially and whenever the object is not occupied by a thread. The *condition* is a Boolean expression that may contain ghost variables.

$$Invariant \rightarrow \mathbf{invariant} \ Condition$$

- Methods: have *MethodSpecClause* which contains *pre* condition and *post* conditions constructs. Permission specifications are defined using *gives*, *takes*, and, *borrow*s. Method permission specifications are used for explicit transfer of permission from one thread to another. A post-condition expression contains variables with apostrophe e.g. if x is the initial value then x' denotes the final

value.

Method → *Access proc*

Name(([ghost] *Direction* *Name* : *Type*)^{*},) (MethodSpecClause)^{*}

MethodSpecClause → **pre Condition**

| **post Condition**

| **takes PermMap**

| **gives PermMap**

| **borrow**s *PermMap*

Interfaces allow declarations, which are fields and methods. Methods are implemented inside the thread of a class implementing the same interface.

2.3.3 Threads

Threads are executable blocks of code, and one block contains a sequence of HARPO commands, local declarations, and other lower-level blocks. For verification purpose, a thread may contain a *claim* specification. A thread of a class executes when a new object of that class is instantiated; threads enable the concurrency which may produce the data races. For one location, the sum of claimed permissions by multiple objects and initial permission must not exceed 1.0 which avoid data races.

Thread may start with a *claim* which claims the locations of an object as its primitive field or array of fields. Sum of claimed permissions for one location must

not exceed 1.0.

$$Thread \rightarrow (\mathbf{thread} [Claim] Block [\mathbf{thread}])$$

$$Claim \rightarrow \mathbf{claim} \ PermMap$$

2.3.4 Statements

- *assert* and *assume*: are written with *condition* which is a boolean expression.

These statements are useful to write the specification of the HARPO program. They can easily let the verifier skip or put assertion on certain code and debugging the large programs.

$$Command \rightarrow \mathbf{assert} \ Condition$$

$$Command \rightarrow \mathbf{assume} \ Condition$$

- Method Call: Methods are called with method call statement and may have *ghost* arguments. Method declarations specify the amount of permission the called thread takes from the calling thread and a pre condition which must be satisfied to make the call.
- Method Implementation: Method implementation is inside the *accept* with correct name of the method to be implementation. It may contain optional *Guard* expression which must be true to get into the method implementation. *Guard* is the only way to protect the locked objects from interference. The role of a *Guard* expression is explained in *with* block.
- Lock: the *with* command is used for implementing locks. Locks are useful while protecting the objects (i.e. locations) from interference and data races. For a

certain location if permission has to be divided around and needs to be unified again for making a particular write operation. A *with* block is used for locking an object and implicitly inferring permissions from object. Sum of permissions for a certain location must not exceed 1.0 in any case.

$$\begin{aligned} \text{Command} \rightarrow & (\textbf{with } \textit{Exp} [\textbf{takes } \textit{PermMap}] [\textbf{Guard}] [\textbf{do}] \textit{Block} \\ & [\textbf{gives } \textit{PermMap}] [\textbf{with}]) \end{aligned}$$

A thread requires the lock object to access shared data. The *with* command is annotated with *takes* and *gives* annotation. When a lock is acquired it *takes* explicitly transfers permission from object to thread and *gives* transfers permission from thread to object after releasing it. For accessibility of data, lock permission and thread permission are added. A guard section is required to check the permission.

- Loops: Loops are much like regular loop structures in higher level programming languages. Loops need loop invariant specifications, where the condition is a boolean expression.

$$\begin{aligned} \text{Command} \rightarrow & \left(\textbf{while } \textit{Expression} [\textbf{Invariant}] [\textbf{do}] \textit{Block} [\textbf{while}] \right) \\ & | \left(\textbf{for } \textit{Name} : \textit{Bounds} [\textbf{Invariant}] [\textbf{do}] \textit{Block} [\textbf{for}] \right) \end{aligned}$$

- Parallel Blocks: Parallel blocks are implemented with *co* command separating the multiple blocks by two vertical bars. Each block may have its own claim. However, the sum of permissions claimed by each block should not exceed the total permission held by the containing thread. For unclaimed locations permissions are kept by the parent thread. At the end of the *co* all permission are

summed up and given back to the containing thread.

$$\begin{aligned} \text{Command} \rightarrow & \left(\text{co } [\text{Claim}] \text{ Block } (|| [\text{Claim}] \text{ Block })^+ [\text{co}] \right) \\ & | \left(\text{co } \text{Name} : \text{Bounds } [\text{do}] [\text{Claim}] \text{ Block } [\text{co}] \right) \end{aligned}$$

2.3.5 Operations

Expression may contain the following operations on *LocSet* and *ObjectId* explained in Section 2.4.

- Readability: *canRead(LocSet)* returns a Boolean which is *true* if *LocSet* is readable, meaning permission on every element of *LocSet* is greater than 0.0; otherwise *false*.
- Writability: *canWrite(LocSet)* : returns a Boolean which is *true* if every element of *LocSet* is writable, meaning permission on *LocSet* must be 1.0 (not more not less); otherwise *false*.
- Accessibility: *permission(ObjectId)* : returns an exact amount of permission the current thread or object has on the location specified by *ObjectId*, a number between 0.0 to 1.0.

2.4 Permission Maps

A permission map contains the mapping of locations to their amount of permissions. Default permission is 1.0. *LocSet* containing number of locations will hold the same full permission for every location inside *LocSet*. One *LocSet* may contains one *ObjectId*

or it may contain set of locations.

$$LocSet \rightarrow ObjectId \mid \{Name : Set [Guard] \text{ do } LocSet\}$$

Name is a bound variable whose value is defined by the *Set* expression. *ObjectId* must be of primitive type.

$$ObjectId \rightarrow Name \mid ObjectId[Expression] \mid ObjectId.Name$$

$$Set \rightarrow \{Expression,..Expression\} \mid \{Expression,..,Expression\} \mid Expression$$

Expressions may contain *ghost* objects. A *Set* has different forms to represent with different notations as follows:

- $\{Expression,..Expression\}$: is inclusive of lower-bound and exclusive of upper-bound.
- $\{Expression,..,Expression\}$: is inclusive of both lower-bound and upper-bound
- *Expression*: is similar to $\{Expression,..Expression\}$

Chapter 3

Boogie Language

Boogie is an intermediate verification language used as a common intermediate representation for verification of higher-level programming languages. Boogie, previously known as BoogiePL, as a language, has imperative and mathematical components. In this chapter, we will discuss the Boogie language syntax and semantics used in the verification of HARPO. Table B.3 in appendix contains all Boogie language keywords¹. This chapter contains information described in Boogie language draft 24 [13].

3.1 Overview

Boogie is a preferred intermediate representation for generating the verification conditions of the modern programming languages. Firstly, a program written in the programming language is translated into Boogie language, i.e., intermediate represen-

¹For differentiation in text, all HARPO keywords are italicized, and Boogie Keywords are bold-faced whereas in production rules both HARPO and Boogie keywords are boldfaced.

tation; secondly, the Boogie code is translated into verification conditions processed with the help of theorem prover *Z3* to validate the verification conditions. A successful proof attempt shows the correctness of the program, and a failed proof attempt indicates possible errors in the program. There are a number of verifiers using Boogie their intermediate representation for verification using the SMT solver *Z3*.

3.2 Declarations

Boogie has seven different kinds of mathematical and imperative declarations. A Boogie program has the following form:

$$\begin{aligned}
 \text{Program} &\longrightarrow \text{Decl}^* \\
 \text{Decl} &\longrightarrow \text{TypeDecl} \mid \text{ConstantDecl} \mid \text{FunctionDecl} \mid \text{AxiomDecl} \mid \text{VarDecl} \mid \\
 &\quad \text{ProcedureDecl} \mid \text{ImplementationDecl}
 \end{aligned}$$

1. Type Declaration: constructs new types.

type *Ref*;

2. Constants: Symbolic constants are declared with constant declaration. *Min-Permission* constant does not have specified value at this point below.

const *Min-Permission*: *Perm*;

3. Function Declaration: creates a mathematical function.

function *isMinPermission*(*Perm*) **returns** (**bool**);

4. Axiom Declaration: postulate the properties of constants and functions. Axioms are postulated with axiom declaration such as below:

```
axiom Min-Permission == 0.0;
```

5. Global variables declaration introduce the mutable variables hold the state on which all procedures operate. e.g.

```
var Permission : PermissionType;
```

6. Procedure Declaration: provide a name to set of execution traces ² which can be constrained with pre-conditions and post-conditions. Set of execution traces are implemented as they are a subset of the procedure declaration as shown below:

```
procedure constructor (This : Ref)
  requires dtype(This) <: Class
  modifies Heap;
{
  //execution traces
}
```

3.3 Boogie Name Space

Boogie has independent namespaces for declared identifiers.

- (a) Types: has a namespace containing global type names (type constructors and type synonyms) and scoped type variables global types names must be

²Execution trace is a non-empty sequence of program states.

distinct from each other, and local types names must be distinct in their scope. Scoped type variables can be shadowed with global type names.

- (b) Constants and Variables: have a namespace containing global variables, constants, and scoped names, i.e., formal parameters, local variables, function names, procedure names and quantified variables.

3.4 Built-in Types

Built-in types are primitive types or bit-vector types.

$$\text{PrimitiveType} \longrightarrow \text{bool} \mid \text{int} \mid \text{real} \mid \text{bv0} \mid \text{bv1} \mid \text{bv2} \dots ,$$

bool denotes the boolean, **int** denotes mathematical integers, and, **real** denotes the mathematical real numbers. In this thesis, we have used built-in types, map types and user defined types.

3.5 User-Defined Types

In addition to built in types, user-defined types declaration are available in Boogie.

$$\text{TypeDecl} \longrightarrow \text{TypeConstructor} \mid \text{TypeSynonym}$$

TypeSynonym is not used in the our experimentation.

3.5.1 Type Constructor

The type constructors in any program must use distinct names and type synonyms.

$$TypeConstructor \rightarrow \text{type } Attribute^* \text{ finite}^? Id Id^*$$

For example, *Permission* is a type constructed from unary type constructor which represents the permissions having some content.

```
type Permission α;
```

Two different instantiations of *Permission* as **real** and **int** representing the permissions of real number and permission of integers respectively. Type constructor must be declared finite otherwise type will have infinite number of individuals. e.g.

```
type finite Permission;  
  
const unique Read-Permission : Permission;  
  
const unique Write-Permission: Permission;  
  
const unique Access-Permission: Permission;  
  
axiom (forall amount : Permission • amount == Read-Permission ∨ amount ==  
       Write-Permission ∨ amount == Access-Permission)
```

3.5.2 Type Synonyms

Type synonym is an abbreviation for the given type. The following declaration rule describes a type synonym where name is first *Id*, type arguments are

remaining *Id*'s and synonym is defined with *Type*.

```
TypeSynonym → type Attribute* Id Id* = Type;
```

We have used *Perm* type synonym using real type definition.

3.6 Axioms

Axioms are declared with the following syntactical description.

```
AxiomDeclaration —> axiom Attribute* Expression;
```

The *Expression* is always a boolean expression not containing any global variables or **old**³ expressions. Axioms contain some properties of the program having constants and functions. Axioms are very useful when certain properties need to be restricted for example, defining the minimum and maximum values of permissions of program locations.

```
axiom MinPermission-Value == 0.0;
```

```
axiom MaxPermission-Value == 1.0;
```

The values of permissions can be bounded using a function below.

```
function IsValidPermission ( Perm ) returns ( bool );
```

```
axiom ( forall x: Perm :: IsValidPermission (x) <==>
```

```
MinPermission-Value <= x && x <= MaxPermission-Value );
```

³Extracting the older value of expression before certain execution to determine the value at prior state.

3.7 Constants and Functions

Constants are syntactically declared using following rule.

ConstantDecl \longrightarrow **const** *Attribute** **unique**? *IdsType OrderSpec*;

IdsType \rightarrow *Id*⁺ : *Type*

Id in *IdsType* represents a symbolic constant having type *Type*. Constant name must be different from other constants and global variables. Declaring a constant with **unique** specifies that the value of constant is different from the values of other **unique** constants having similar type. Applying **unique** to declarations in 3.5.1 asserts that the following condition is *true*.

axiom *Read-Permission* \neq *Write-Permission* \wedge *Write-Permission* \neq
Access-Permission \wedge *Access-Permission* \neq *Read-Permission*

Function are declared as:

FunctionDecl \rightarrow **function** *Attribute** *Id FuncSig*; |

function *Attribute** *Id FuncSig* { *Expression* }

FuncSig \rightarrow *TypeArgs*? **returns** (*FuncArg*)

FuncArg \rightarrow *FuncArgName*? *Type*

FuncArgName \rightarrow *Id*:

e.g.

```

function isInt32(int) returns(bool);

axiom (forall x : int :: isInt32(x)  $\iff$  minInt32  $\leq$  x  $\&\&$  x  $\leq$ 
maxInt32);

```

Where *minInt32* and *maxInt32* are symbolic constants.

3.8 Expressions

Boogie expressions consist of symbolic constants, global variables, equality and arithmetic relational operators, logical quantifications, and ordering operators. Expressions in Boogie are much like expressions defined in a high-level programming language, which makes them readable and explanatory by themselves. However, expressions should not be loosely typed, e.g., \vee and \wedge have the same binding power and can not both occur in the same expression without intervening parentheses, so an expression containing both must be parenthesized to avoid ambiguity. Also, for division and modulo, they have different definitions in source languages, Boogie provides the syntax of both ‘/’ and ‘%’ respectively, but left their definition in axiomatization for the programmer. Boogie expression is constructed with a combination of the following operators. [13]

EquivalentOp → $\iff|<==>$

ImplicationOp → $\implies|==>$

OrOp → $\vee \mid \parallel$

AndOp → $\wedge \mid \&\&$

EqualOp → $=$

NotEqualOp → $\neq \mid !=$

GreaterThanOp → $>$

LessThanOp → $<$

LessThanOrEuqalOp → $\leq \mid <=$

GreaterThanOrEqualOp → $\geq \mid >=$

ParentTypeOp → $<:$

ConcatinationOp → $++$

AdditionOp → $+ \mid -$

MultiplicationOp → $* \mid / \mid \%$

UnaryOp → $\neg \mid ! \mid -$

QuantificationOp → **forall** \mid **exists** $\mid \forall \mid \exists$

QuantificationSeparatporOp → $\bullet \mid ::$

3.9 Procedures

Procedures have caller and callee traces determined by the procedure specifications and signature defined in procedure declaration. Implementation of the procedure also defines the set of execution traces. Procedures are invoked by the call statement, which makes the trace of the caller. The correct implementation of the procedure makes the execution traces subset of procedure definition. The procedure declaration has two different syntactic forms.

```
ProcedureDecl —> procedure Attribute* Id ProcSignature ProcSpecClause*
| procedure Attribute* Id ProcSignature ProcSpecClause* Body
ProcSignature —> TypeArg? OutParams? (IdsTypeWhere,*)  

OutParams —> returns (IdsTypeWhere,*)
```

ProcSignature contains the type arguments as *in* parameters and *out* parameters. *Id's* and type arguments must be distinct from each other. A procedure specification clause contains more clauses; and boolean expressions such as pre-conditions, post-conditions, and modification clause. Caller's role is to establish required pre-condition to hold, and implementation will assume that the precondition is in effect. Modification clause enlists the global variables allowed to change inside the implementation. Post-conditions are generally relate to the execution traces of before and after the execution of the implementation of the

same procedure.

SpecClaus —> **requires** *Expression*;

modifies *Id*^{*};

ensures *Expression*;

3.10 Assertions and Assumptions

The assert and assume statements are used to accept or pass the conditions that check the correctness and feasibility of the traces, respectively. For example, a HARPO assignment:

obj *a* : **Int32** := 5 + 9;

will be translated to Boogie as

assert *isInt32(9 + 5)*; *or assert* *isInt32(14)*;

An assertion is very essential to make strong reasoning on traces leading to errors, e.g., checking the bounds of array and null references, etc. These assertions provide the map of failure trace used by source language to spot erroneous code.

Assumptions are used to render executions traces which are not feasible and may cause to a failed assumption. For example, to check that the heap satisfies a property after the assignment, one would use “assert” and to check that the heap obeys a property after the assignment, one would use “assume”. The well-definedness of the heap after every operation performed on the heap as shown

below:

```
assert isHeapDefined(Heap);  
  
Heap[This.Math, Math.count] := Heap[This.Math, Math.count] + 1;  
  
assume isHeapDefined(Heap);  
  
axiom ( $\forall$  Heap : HeapType • isHeapDefined(Heap)  $\Rightarrow$  ... );
```

3.11 Havoc

Havoc statement is used to assign arbitrary values to mutable variable or set of mutable variables. Values assigned to variables respect the axioms and **where** clause of the mutable variable.

```
var a : real where a >= 0.0;  
  
var b : real where a >= b >= 0.0;  
  
assume isHeapDefined(Heap)  
  
havoc a , b ;
```

Here **havoc** will set a as positive real number and b as positive real number but less than or equal to a .

3.12 Control Transfer

Boogie uses a **goto** statement for transfer of control to certain labeled points in a program. Labels need to be distinct inside the same implementation body. The

return statement terminates the execution trace that approaches it. Implicit return is always located at the end of the implementation.

3.13 Conditional Blocks and Loops

Boogie has two different semantics of **if** statement; one with *WildCardExpression*, and other with boolean expression, which determine the **if** block or **else** block to be executed. For arbitrarily choosing the alternatives *WildCardCharacter* (*) is used. 3.1 has *WildCardExpression* as a boolean expression to decided *Then* execution or *Els* execution. For 3.2 it decided arbitrarily between *Fst Block* or *Snd Block*.

if(*WildCardExpression*) *Then* **else** *Els* (3.1)

if(*) *Fst Block* **else** *Snd Block* (3.2)

Chapter 4

HARPO to Boogie Translation

Translation of the HARPO program into Boogie consists of two types of code that are combined to get a complete translated version. One part is static; it is independent of the HARPO source program and is called the Boogie prelude. The second part is customized and generated from the HARPO code under verification. Section 4.1 contains the description of the prelude as a boilerplate of customized translation. Section 4.2 provides a description of how HARPO program components are translated into their Boogie counterparts. The seminal work on HARPO to Boogie translation is completed in the thesis [12]. In this chapter, we review and update the translation according to up to date HARPO language syntax. The translation is also improved based on the implementation of dependencies. Following procedure is followed during the translation and verification:

- Translating the HARPO program specifications into an intermediate veri-

fication language (IVL), Boogie.

- Boogie source is converted into verification conditions and checked by the Boogie verifier, which generates the error report.
- An error report processor processes the error report and updates the error recorder.
- Verification errors are mapped back to HARPO source code (line numbers) in the error recorder.

4.1 Boogie Prelude

Boogie prelude contains the declaration of types, constants, axioms, and functions accessed by customized translated code. The prelude incorporates the basic properties required for all HARPO programs.

4.1.1 Class and Interface Type

HARPO class and interface names are modeled as members of a Boogie type. Class or interface declarations result in constants of type *ClassName*. To get the type of particular object *Ref* the function *dtype* is used.

```
type ClassName;  
  
function dtype(Ref) returns(ClassName);
```

4.1.2 Global Variables

The translation methodology uses a heap memory model to represent the locations, arrays, and objects of a HARPO program.

- (a) Heap Type: for memory modeling, a heterogeneous heap memory model is used that maps object references and fields to their values where result type is dependent on the type of input.

```
type Ref;  
type Field x;  
type HeapType = <x>[Ref, Field x] x;  
var Heap : HeapType;
```

The above declaration implies the semantics of expression where the value depends on the type of result:

forall x:: (Ref, Field x) -> x

- (b) Array Heap Type: arrays are modeled with ArrayHeap, which provides concise specifications and quantifications. Changes in Heap and Array Heap are independent.

```
type ArrayRef x;  
type ArrayHeapType = <x>[ArrayRef x, int] x;  
var ArrayHeap : ArrayHeapType;
```

- (c) Perm Type: **Perm** type is a shadow of the Boogie **real** type used to bound the permission value between 0.0 and 1.0.

```
type Perm = real;  
const unique minPer : Perm;  
const unique maxPer : Perm;
```

- (d) Permission Type: *PermissionType* is used to create Permission variable which hold permission values of *HeapType* variable.

```
type PermissionType : <x>[Ref, Field x] Perm;  
var Permission : PermissionType;
```

- (e) Array Permission Type: *ArrayPermissionType* is similar to *PermissionType* it holds the permission of *ArrayHeapType* variables.

```
type ArrayPermissionType : <x>[ArrayRefx, int] Perm;  
var ArrayPermission : ArrayPermissionType;
```

4.1.3 Axioms and Functions

- Integers

Section 2.2.2 contains various HARPO integer types and the range of their values. All integer types in HARPO are translated to **int** type in Boogie. Functions are used to maintain the range of values for all integer type constants in Boogie.

```
const unique minInt8: int;  
axiom minInt8 == -128;  
const unique maxInt8: int;  
axiom maxInt8 == 127;
```

```
const unique minInt16: int;  
axiom minInt16 == -32768;  
const unique maxInt16: int;  
axiom maxInt16 == 32767;
```

```
const unique minInt32: int;  
axiom minInt32 == -2147483648;  
const unique maxInt32: int;  
axiom maxInt32 == 2147483647;
```

```
const unique minInt64: int;  
axiom minInt64 == -9223372036854775808;  
const unique maxInt64: int;  
axiom maxInt64 == 9223372036854775807;
```

```
function isInt8(int) returns (bool);  
axiom (forall x:int::isInt8(x)<==>minInt8<= x && x <= maxInt8);
```

```
function isInt16(int) returns (bool);
```

```
axiom (forall x:int::isInt16(x)<==>minInt16<= x && x <= maxInt16);
```

```
function isInt32(int) returns (bool);
```

```
axiom (forall x:int::isInt32(x)<==>minInt32<= x && x <= maxInt32);
```

```
function isInt64(int) returns (bool);
```

```
axiom (forall x:int::isInt64(x)<==>minInt64<= x && x <= maxInt64);
```

- Real

All real types in HARPO are converted into the **real** type in Boogie.

The translation of the real types restricts the value inside inside allowed range. Section 2.2.2 contains all the real types available in HARPO and their proper bounds. For the representation of large real numbers, we divided them into smaller symbolic constants and used them in **axioms** for building boolean expressions. For instance *TenPow16* is a unique value containing 10.0×10^{16} and so on.

```
// 6.10 E - 5
```

```
const unique minReal16: real;
```

```
axiom minReal16 == -6.10/10000.0;
```

```
// 6.55 E + 4
```

```
const unique maxReal16: real;
```

```
axiom maxReal16 == 6.55*10000.0;
```

```

// 3.402823466 E - 38

const unique minReal32: real;

axiom minReal32 == -3.402823466/(TenPow32*10000.0);

//3.402823465 E + 38

const unique maxReal32: real;

axiom maxReal32 == 3.402823465 * (TenPow32*10000.0);

// -1.7976931348623157 E - 308

const unique minReal64: real;

axiom minReal64 == -1.7976931348623157 / (TenPow64*TenPow64*TenPow64*
→ TenPow64*TenPow32*TenPow16*1000.0);

// 1.7976931348623156 E + 308

const unique maxReal64: real;

axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64*TenPow64*
→ TenPow64*TenPow32*TenPow16*1000.0);

function isReal16(real) returns (bool);

axiom (forall x:real::isReal16(x)<==>minReal16 <= x && x <= maxReal16);

function isReal32(real) returns (bool);

axiom (forall x:real::isReal32(x)<==> minReal32 <= x && x <= maxReal32);

```

```

function isReal64(real) returns (bool);
axiom (forall x:real::isReal64(x) $\leqslant\geqslant$  minReal64  $\leqslant\leqslant$  x  $\&\&$  x  $\leqslant\leqslant$  maxReal64);

```

- Perm

The idea of permission transfer in HARPO was introduced in 2017 [32].

The minimum and maximum permission amount on a location are 0.0 and 1.0, respectively. In order to restrict valid permission amount axioms and functions are used.

```

const unique minPerAmount : Perm;
axiom minPerAmount == 0.0;
const unique maxPerAmount : Perm;
axiom maxPerAmount == 1.0;

```

4.2 Customized Translation

HARPO programs are written in an object-oriented fashion. Translation of the source program is dependent on the abstract syntax tree generated from HARPO source program. Abstract Syntax Tree (AST) generation and manipulation for HARPO programs explained in [33]. All syntactical errors are reported while producing the AST. Since HARPO programs are modular in nature, whereas the Boogie code is executed sequentially. This difference produced some translation dependencies. This section explains only the translation of HARPO annotations, declarations, commands, and, expressions into their Boogie counterparts. For simplicity only parts of HARPO programs are trans-

lated in following sub-sections. Complete translations are provided later in this chapter. Table B.1 in appendix shows primitives types mapping from HARPO to Boogie.

4.2.1 Class and Interface Translation

Classes and Interfaces are translated into constants in Boogie. For subtyping in Boogie ‘`<:`’ is used.

```
(interface Arithmetic interface)
(class Math() implements Arithmetic

/* Fields and Methods */

class)
```

HARPO Code

```
/*Boogie prelude*/
const unique Arithmetic : ClassName;
const unique Math : ClassName <: Arithmetic;
```

Boogie Translation

4.2.2 Fields and Constants Translation

The fields of HARPO are translated into constants in Boogie and initialized in constructor procedure. Constants translated to Boogie constants and their values are defined using an independent axiom each. The translation of identifiers prepends the name with class names where ‘.’ is only part of an identifier.

```
(class Math()
  obj count : Int32 := 0;
  const maxStep : Int8 := 10;
class)
```

HARPO Code

```
/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.count: Field int;
const unique Math.maxStep: int;
axiom Math.maxStep == 10;

procedure Math.constructor(This.Math : Ref)
  requires dtype(This.Math) <: Math;
  modifies Heap;
{
  Heap[This.Math, Math.count] := 0;
}
```

Boogie Translation

4.2.3 Class Invariant Translation

A Class invariant is always asserted in the constructor procedure of the class.

```
(class Math()
  invariant x >_ 0;
  obj x : Int32 := 0;
class)
```

HARPO Code

```
/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.x: Field int;
procedure Math.constructor(This.Math: Ref)
```

```

requires dtype(This.Math) <: Math;
modifies Heap;
{
    var Permission : PermissionType;
    //The invariant reads x asserts a nonzero permission on x
    assert Permission[This.Math,Math.x] > 0.0;
    Heap[This.Math,Math.x] := 0;
    assert Heap[This.Math,Math.x] >= 0;
}

```

Boogie Translation

4.2.4 Class Claim Translation

Class claim adds permission amount to a permission variable in the constructor procedure.

```

(class Math()
  claim x@1.0;
  invariant x >_ 0;
  obj x : Int32 := 0;
class)

```

HARPO Code

```

/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.x: Field int;
procedure Math.constructor(This.Math:Ref)
  requires dtype(This.Math) <: Math;
  modifies Heap;
{
    var Permission : PermissionType;
    Permission[This.Math,Math.x] := 1.0;
    assert Permission[This.Math,Math.x] > 0.0;
    Heap[This.Math,Math.x] := 0;
    assert Heap[This.Math,Math.x] >= 0;
}

```

Boogie Translation

4.2.5 Class Constructor Translation

The class constructor is translated into a constructor procedure. Constructor arguments are fields and so are represented by constants in Boogie. Pre-condition(s) and post-conditions of the class constructor are assumed and asserted respectively inside the constructor procedure of Boogie.

```
(class Math (in offSet: Int32)
  pre offSet >0;
  post count < 999;

  claim count; // default claim amount is '1.0'
  invariant count >_0 /\canRead(offSet);

  obj count:Int32:=0;

class)
```

HARPO Code

```
const unique Math: ClassName;
const unique Math.count: Field int;
const unique Math.offSet: Field int;
procedure Math.constructor(This.Math:Ref)
  requires dtype(This.Math)<: Math;
  modifies Heap;
{
  var Permission: PermissionType;
  //Pre Condition
  assume Heap[This.Math,Math.offSet]>0;

  //Add permissions to input parameters
  Permission[This.Math,Math.offSet]:= 1.0;

  //Claim
  Permission[This.Math,Math.count]:= 1.0;

  //Initialization
  Heap[This.Math,Math.count]:= 0;
```

```
// Class Invariant
assert Heap[This.Math.Math.count] >= 0 && Permission[This.
    Math.Math.offset] > 0.0;
// Post Condition
assert Heap[This.Math.Math.count] < 999;
}
```

Boogie Translation

4.2.6 Thread Translation

Threads are blocks of the code declared inside a class, and it may contain statements and method implementations. Threads may claim permissions on locations and can pass on permissions to each other. Threads are translated to stand-alone procedures in Boogie. The total permission on a heap location is always asserted to be any real number between 0.0 and 1.0.

```
(class Math (in offset: Int32)
  pre offset > 0;
  post count < 999
  invariant count > -0 / \canRead(offset);
  obj count:Int32:=0;
  (thread (*t0*) claim count, offset
   (accept reset()
    count := offset;
   accept)
  thread)
 class)
```

HARPO Code

```

/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.count: Field int;
const unique Math.offSet: Field int;
procedure Math.constructor(This.Math:Ref)
  requires dtype(This.Math) <: Math;
  modifies Heap;
{
  var Permission : PermissionType;
  //Add permission to input parameter
  Permission [This.Math,Math.offSet] := 1.0;
  //Pre Condition
  assume Heap [This.Math,Math.offSet] >0;
  //Initialization
  Heap [This.Math,Math.count] := 0;
  //Class Invariant
  assert Heap [This.Math,Math.count] >= 0 && Permission [This.Math,Math.offSet] >0.0;
  //Post Condition
  assert Heap [This.Math,Math.count] < 999;
}

procedure Math.t0(This.Math: Ref)
  requires dtype(This.Math)<: Math;
  modifies Heap;
{
  var old_Heap , temp_Heap : HeapType;
  var localPermission: PermissionType;
  //Thread Claim
  havoc localPermission;
  localPermission [This.Math,Math.count] := 1.0;
  localPermission [This.Math,Math.offSet] := 1.0;
  /*Accept Command Translation*/
}

```

Boogie Translation

4.2.7 Method Translation

A method consists of the method declaration, method implementation, and method call. A thread may contain method implementation and method calls.

The claim specification of the thread declares the initial permission on the locations accessed in the thread body. The declaration and implementation of a method are combined in translation. However, each method call is translated independently into Boogie.

```
(class Math (in offSet: Int32)
  pre offSet >0;
  post count < 999
  invariant count >_0 /\ canRead(offSet);
  obj count:Int32:=0;
  proc increment()
    pre count >_0
    post count = count + 1;
  (thread (*t0*) claim count, offSet
    (accept increment()
      count := count + 1;
      accept)
    thread)
  class)
```

HARPO Code

```
/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.count: Field int;
const unique Math.offSet: Field int;
procedure Math.constructor(This.Math:Ref)
  requires dtype(This.Math) <: Math;
  modifies Heap;
{
  var Permission : PermissionType;
  Permission[This.Math,Math.offSet] := 1.0;
  assume Heap[This.Math,Math.offSet]>0;
  Heap[This.Math,Math.count] := 0;
  assert Heap[This.Math,Math.count] >= 0 && Permission[This.Math,Math.offSet]>0.0;
  assert Heap[This.Math,Math.count] < 999;
}
procedure Math.t0(This.Math:Ref)
  requires dtype(This.Math) <: Math;
```

```

modifies Heap;
{
    var localPermission : PermissionType;
    var oldHeap : HeapType;
    //Thread Claim
    havoc localPermission;
    localPermission [This.Math.Math.count] := 1.0;
    localPermission [This.Math.Math.offSet] := 1.0;

    goto increment;
    increment:
    //Assume pre-condition
    assert localPermission [This.Math.Math.count] > 0.0;
    assume Heap [This.Math.Math.count] > 0;
    assert localPermission [This.Math.Math.count] == 1.0;
    havoc Heap;
    oldHeap := Heap;
    Heap [This.Math.Math.count] := Heap [This.Math.Math.count] + 1;
    assert Heap [This.Math.Math.count] == oldHeap [This.Math.Math.count] + 1;
}

```

Boogie Translation

4.2.8 Expression Translation

Expressions are translated into two steps. First, check the definedness of expressions; the second translates into their Boogie counterparts. The function *isDefined(Arg)* is generic representation for numerous function defined in the Boogie prelude for checking the definedness of expression values as shown in Table 4.1. Table B.4 in appendix shows one-to-one mapping of HARPO to Boogie operators. Table 4.2 has HARPO to Boogie expressions translation.

Integer	Floating Point	Boolean	Permission Value
$isInt8(arg)$	$isReal16(arg)$	$isBool(arg)$	$isValidPermission(arg)$
$isInt16(arg)$	$isReal32(arg)$		
$isInt32(arg)$	$isReal64(arg)$		
$isInt64(arg)$			

Table 4.1: User Defined Functions for Checking Defineness

Expression	HARPO	Boogie
$x @ y$, with @ being $+, -, *, /$, div	$x @ y$	$isDefined(Heap[This.Math.Math.x]);$ $isDefined(Heap[This.Math.Math.x]);$ $Heap[This.Math.Math.x] @ Heap[This.Math.Math.y]$
$x \odot y$, with \odot being $<, >, <, >, ==, !=$	$x \odot y$	$isDefined(Heap[This.Math.Math.x]);$ $isDefined(Heap[This.Math.Math.x]);$ $Heap[This.Math.Math.x] \odot Heap[This.Math.Math.y]$
$x @ y$ with $@$ being and , $/\wedge$, or , $/\vee$	$x @ y$	$isDefined(Heap[This.Math.Math.x]);$ $isDefined(Heap[This.Math.Math.x]);$ $Heap[This.Math.Math.x] @ Heap[This.Math.Math.y]$
Φx , with Φ being $-$, \sim ,	Φx	$isDefined(Heap[This.Math.Math.x]); !Heap[This.Math.Math.x];$
$x \dagger y$, with \dagger being $==>, <==, <==>$	$x \dagger y$	$isDefined(Heap[This.Math.Math.x]);$ $isDefined(Heap[This.Math.Math.x]);$ $Heap[This.Math.Math.x] \dagger Heap[This.Math.Math.y]$
Read Permission	$canRead(x)$	$isDefined(Heap[This.Math.Math.x]);$ $Permission[This.Math.Math.x] > 0.0$
Write Permission	$canWrite(x)$	$isDefined(Heap[This.Math.Math.x]);$ $Permission[This.Math.Math.x] == 0.0$
Permission Value	$permission(x)$	$isDefined(Heap[This.Math.Math.x]); Permission Value$

Table 4.2: Expression Translation

4.2.9 Command Translation

Command translation in this section contains the translation of the HARPO commands, assuming the prelude, required constants, constructor procedures, and thread procedures are available.

4.2.9.1 Skip Command Translation

Skip command translation is ignored during translation into the Boogie code generation.

4.2.9.2 Sequence Command Translation

The sequence Command is used to break down the block into two segments of commands named *fst-command* and *snd-command*. The *fst-command* is the first command in the sequence, and *snd-command* is the rest of commands inside the thread body that may contain another sequence of commands.

```
(class CmdContainer
  (thread(*t0*)
   / fst-command
   / snd-command
  thread)
 class)
```

HARPO Code

```
/*Boogie prelude*/
const unique CmdContainer: ClassName;
/*Constructor Procedure*/
procedure CmdContainer.t0(This.CmdContainer: Ref)
```

```
{  
    // First Command Translation  
    // Second Command Translation  
}
```

Boogie Translation

4.2.9.3 Local Declaration Command

A local declaration is inside the thread body. All local declarations are promoted to constant fields in Boogie. The new declaration is initialized with assignment command. Before the assignment, the expression value is checked for the definedness of initialization value.

```
(class CmdContainer()  
(thread(*t0*)  
  obj x: Int32 := 0;  
  obj y : Real32 := 0.0;  
  thread)  
class)
```

HARPO Code

```
/*Boogie prelude*/  
const unique CmdContainer: ClassName;  
const unique CmdContainer.t0.x : Field int;  
const unique CmdContainer.t0.y : Field real;  
procedure CmdContainer.constructor(This.CmdContainer:Ref)  
requires dtype(This.CmdContainer) <: CmdContainer;  
modifies Heap;  
{  
  var local_Permission: PermissionType;  
  havoc local_Permission;  
  local_Permission [This.CmdContainer, CmdContainer.t0.x] :=  
    1.0;  
  assert isInt32(0);
```

```

Heap[This.CmdContainer, CmdContainer.t0.x] := 0;
assert isValidPermission(1.0);
local_Permission[This.CmdContainer, CmdContainer.t0.x] :=
    1.0;
assert isReal32(0.0);
Heap[This.CmdContainer, CmdContainer.t0.y] := 0.0;
//Remove Permisssion
local_Permission[This.CmdContainer, CmdContainer.t0.x] := 0.0;
local_Permission[This.CmdContainer, CmdContainer.t0.y] := 0.0;
}

```

Boogie Translation

4.2.9.4 Assignment Command

Assignment command allows expressions on the right-hand side of the assignment. Fields can not be assigned with reference types.

```

(class CmdContainer()
  obj x : Int32 := 0;
  (thread(*t0*)
    x := x+1;
  thread)
class)

```

HARPO Code

```

/*Boogie prelude*/
const unique CmdContainer: ClassName;
const unique CmdContainer.x : Field int;
/*Constructor Procedure*/
procedure CmdContainer.t0(This.CmdContainer: Ref)
  requires dtype(This.CmdContainer) <: CmdContainer;
  modifies Heap;
{
  var local_Permission: PermissionType;
  var oldHeap : HeapType;
  havoc oldHeap, local_Permission;
  assert local_Permission[This.CmdContainer, CmdContainer.x] >
    0.0;

```

```

assert local_Permission [This . CmdContainer , CmdContainer . x] ==
    1.0;
assert isInt32 (Heap [ This . CmdContainer , CmdContainer . x]) &&
    isInt32 (Heap [ This . CmdContainer , CmdContainer . x]) &&
    isInt32 (1);
oldHeap := Heap;
Heap [ This . CmdContainer , CmdContainer . x] := Heap [ This .
    CmdContainer , CmdContainer . x] + 1;
}

```

Boogie Translation

4.2.9.5 If Command

The *if* command translation contains the check for definedness of the guard expression. Then the HARPO *if* command is converted into a Boogie **if** command. The thread must check that it has enough permission on guard expression. Then, it is translated into Boogie expression, according to Table 4.2.

```

(class CmdContainer()
  obj x : Int32 := 0;
  obj flag : bool := false;
  (thread(*t0*) claim x
    if(flag = true)
      x := x + 1;
    else if (flag = false)
      x := x - 1;
  thread)
class)

```

HARPO Code

```

/*Boogie prelude*/
const unique CmdContainer: ClassName;
const unique CmdContainer.x : Field int;
const unique CmdContainer.flag : Field bool;

```

```

/*Constructor Procedure*/
procedure CmdContainer.t0(This.CmdContainer : Ref)
  requires dtype(This.CmdContainer) <: CmdContainer;
  modifies Heap;
{
  var Permission: PermissionType;
  var oldHeap : HeapType;
  havoc oldHeap, Permission;
  Permission [This.CmdContainer, CmdContainer.x] := 1.0;
  //Assume initial values of claimed locations
  assume Heap[This.CmdContainer, CmdContainer.x] == 0;
  assert Permission[This.CmdContainer, CmdContainer.x] > 0.0;
  //Check definedness of guard expression
  assert isBool(Heap[This.CmdContainer, CmdContainer.flag]) &&
    isBool(true);
  if(Heap[This.CmdContainer, CmdContainer.flag] == true)
  {
    assert Permission[This.CmdContainer, CmdContainer.x] > 0.0;
    assert Permission[This.CmdContainer, CmdContainer.x] == 1.0;
    assert isInt32(Heap[This.CmdContainer, CmdContainer.x]) &&
      isInt32(Heap[This.CmdContainer, CmdContainer.x]) &&
      isInt32(1);
    oldHeap := Heap;
    Heap[This.CmdContainer, CmdContainer.x] := Heap[This.
      CmdContainer, CmdContainer.x] + 1;
  }
  else if(Heap[This.CmdContainer, CmdContainer.flag] == false)
  {
    assert Permission[This.CmdContainer, CmdContainer.x] > 0.0;
    assert Permission[This.CmdContainer, CmdContainer.x] == 1.0;
    assert isInt32(Heap[This.CmdContainer, CmdContainer.x]) &&
      isInt32(Heap[This.CmdContainer, CmdContainer.x]) &&
      isInt32(1);
    oldHeap := Heap;
    Heap[This.CmdContainer, CmdContainer.x] := Heap[This.
      CmdContainer, CmdContainer.x] - 1;
  }
}

```

Boogie Translation

4.2.9.6 While Command

The HARPO's *while* command is translated into Boogie **while** command. This translation contains the copy of the old heap upon entering into the loop, then the invariant is checked for definedness and translated into the Boogie loop invariant. The guard expression is checked for its definedness, and another loop invariant is applied to ensure the guard's definedness during loop execution. The unused locations in a heap must remain unchanged, which is also achieved by another loop invariant.

```
(class CmdContainer()
  obj count : Int32 := 0;
  (thread(*t0*) claim count
    while(count <_ 999) invariant count >_ 0
      count := count + 1;
    thread)
  class)
```

HARPO Code

```
/*Boogie prelude*/
const unique CmdContainer: ClassName;
const unique CmdContainer.count : Field int;

/*Constructor Procedure*/
procedure CmdContainer.t0(This.CmdContainer : Ref)
  requires dtype(This.CmdContainer) <: CmdContainer;
  modifies Heap;
{
  var Permission: PermissionType;
  var oldHeap: HeapType;
  havoc oldHeap, Permission;
  Permission[This.CmdContainer, CmdContainer.count] := 1.0;

  //Assume initial values of claimed locations
  assume Heap[This.CmdContainer, CmdContainer.count] == 0;
```

```

assert Permission [ This . CmdContainer , CmdContainer . count ] >
    0.0;
oldHeap := Heap;
while (Heap [ This . CmdContainer , CmdContainer . count ] <= 999)
invariant (forallx r:Ref, f: Field x :: !(r == This .
    CmdContainer && f == CmdContainer . count ) ==> Heap [ r , f ]
    == oldHeap [ r , f ]) ;
// Invariant containing definedness of guard expression
invariant (isInt32 (Heap [ This . CmdContainer , CmdContainer . count
]) && isInt32 (999) && Heap [ This . CmdContainer ,
CmdContainer . count ] <=999);
{
//Unchanged Heap
assert Permission [ This . CmdContainer , CmdContainer . count ] >
    0.0;
assert Permission [ This . CmdContainer , CmdContainer . count ] ==
    1.0;
assert isInt32 (Heap [ This . CmdContainer , CmdContainer . count ])
    && isInt32 (Heap [ This . CmdContainer , CmdContainer . count ])
    && isInt32 (1);
oldHeap := Heap;
Heap [ This . CmdContainer , CmdContainer . count ] := Heap [ This .
    CmdContainer , CmdContainer . count ] + 1;
}
}

```

Boogie Translation

4.2.9.7 For Command

HARPO *for* loops are translated into **while** loops in Boogie. For instance, consider an example of accumulating the sum of a counter variable in for loop. The source code given below has the translation into a while loop in Boogie. This example below contains ‘Accept’ command inside to body of ‘For’ to explain the implementation of thread synchronization translated into **goto** command in Boogie. For command below has accept command inside and translated

accordingly. When it accepts multiple procedures implementation, then **goto** command enlists all the procedures separating by commas, and **goto** command arbitrarily picks up the one implementation. At the end of one implementation, it should send the execution control to the **end** label.

```
(class CmdContainer()
  procedure Add(in n : Int32)
    takes sum@1.0
    pre n >= 0 && sum >= 0
    post sum' = sum + n
    gives sum@1.0
    obj sum : Int32 := 0
    (thread (*t0*)
      (while true
        (accept add(in n : Int32)
          (for i : n
            invariant acc i@1.0 && (0 _< i && i < n)
            invariant acc sum@1.0 && sum = i
            sum := sum + 1
            for)
          accept)
        while)
      thread)
    class)
```

HARPO Code

```
/*Boogie prelude*/
const unique CmdContainer: ClassName;
const unique CmdContainer.sum : Field int;
const unique CmdContainer.add.n : Field int;
const unique CmdContainer.t0.forVar.i: Field int;

/*Constructor Procedure*/
procedure CmdContainder.constructor(This.CmdContainer: Ref)
  requires dtype(This.CmdContainer) <: CmdContainer;
  modifies Heap;
{
  var Permission : PermissionType;
  havoc Heap;
```

```

Permission [ This . CmdContainer , CmdContainer .sum] := 1.0;
Heap [ This . CmdContainer , CmdContainer .sum] := 0;
}
procedure CmdContainer . t0 ( This . CmdContainer : Ref)
  modifies Heap ;
  requires dtype( This . CmdContainer ) <: CmdContainer ;
{
  var Permission , oldPermission : PermissionType ;
  var oldHeap : HeapType ;
  havoc Heap , Permission , oldPermission ;
  oldHeap := Heap ;
  //While loop
  while( true )
    invariant ( forall<x> r:Ref , f: Field x :: !(r == This .
      CmdContainer && f == CmdContainer .sum) ==> 0.0 <=
      Permission [ r , f ] && Permission [ r , f ] <= 1.0 ) ;
  {
    //Accept Command
    goto Add;
    Add:
    //Takes: add permissions
    Permission [ This . CmdContainer , CmdContainer .sum] := 1.0 ;
    //Assume Pre Condition
    assert Permission [ This . CmdContainer , CmdContainer .add .n] >
      0.0 && Permission [ This . CmdContainer , CmdContainer .sum]
      > 0.0 ;
    assume Heap [ This . CmdContainer , CmdContainer .add .n] >= 0 &&
      Heap [ This . CmdContainer , CmdContainer .sum] >= 0 ;
    //For loop to While loop
    while( Heap [ This . CmdContainer , CmdContainer .t0 .forVar .i ] <
      Heap [ This . CmdContainer , CmdContainer .add .n] )
      invariant Permission [ This . CmdContainer , CmdContainer .t0 .
        forVar .i ] = 1.0 && 0 <= Heap [ This . CmdContainer ,
        CmdContainer .t0 .forVar .i ] && Heap [ This . CmdContainer ,
        CmdContainer .t0 .forVar .i ] < Heap [ This . CmdContainer ,
        CmdContainer .add .n] ;
      invariant Permission [ This . CmdContainer , CmdContainer .sum]
      = 1.0 && Heap [ This . CmdContainer , CmdContainer .sum] ==
        Heap [ This . CmdContainer , CmdContainer .t0 .forVar .i ] ;
      invariant ( forall<x> r:Ref , f: Field x :: !(r == This .
        CmdContainer && f == CmdContainer .sum) ==> Heap [ r , f ]
        = oldHeap [ r , f ] ) ;
    {
      assert Permission [ This . CmdContainer , CmdContainer .sum] ==
        1.0 ;

```

```

Heap[ This . CmdContainer , CmdContainer . sum ] := Heap[ This .
    CmdContainer , CmdContainer . sum ] + 1;
assert isInt32(Heap[ This . CmdContainer , CmdContainer . sum ]); 
Heap[ This . CmdContainer , CmdContainer . t0 . forVar . i ] := Heap[ 
    This . CmdContainer , CmdContainer . t0 . forVar . i ] + 1;
assert isInt32(Heap[ This . CmdContainer , CmdContainer . t0 .
    forVar . i ]); 
//post condition
assert Heap[ This . CmdContainer , CmdContainer . sum ] > 0;
assert Heap[ This . CmdContainer , CmdContainer . sum ] ==
    oldHeap[ This . CmdContainer , CmdContainer . sum ] + 1;
//Gives: remove permissions
assert Permission[ This . CmdContainer , CmdContainer . sum ] ==
    1.0;
Permission[ This . CmdContainer , CmdContainer . sum ] := 0.0;
assert Permission[ This . CmdContainer , CmdContainer . sum ] ==
    oldPermission[ This . CmdContainer , CmdContainer . sum ] -
    1.0;
goto end;
end:
}
}
}

```

Boogie Translation

4.2.9.8 Assert and Assume Command

The ‘*assert*’ and ‘*assume*’ commands in HARPO have their counterparts in Boogie. Each HARPO *assert* command is translated into a Boogie **assert** command and HARPO *assume* command translated in Boogie **assume** command.

```

(class CmdContainer()
(thread(*t0*)
obj x: Int32 := 0;
obj y : Real32 := 0.0;
assume x = 0;
assert x = y;

```

thread)
class)

HARPO Code

```
/*Boogie prelude*/
const unique CmdContainer: ClassName;
const unique CmdContainer.t0.x : Field int;
const unique CmdContainer.t0.y : Field real;

procedure CmdContainer.t0( This.CmdContainer : Ref)
requires dtype(This.CmdContainer) <: CmdContainer;
modifies Heap;
{
    var local_Permission: PermissionType;
    havoc local_Permission;
    local_Permission[This.CmdContainer, CmdContainer.t0.x] := 1.0;
    assert isInt32(0);
    Heap[This.CmdContainer, CmdContainer.t0.x] := 0;
    local_Permission[This.CmdContainer, CmdContainer.t0.x] := 1.0;
    assert isReal32(1.0);
    Heap[This.CmdContainer, CmdContainer.t0.y] := 0.0;
    assert local_Permission[This.CmdContainer, CmdContainer.t0.x] > 0.0;
    assume Heap[This.CmdContainer, CmdContainer.t0.x] == 0;
    assert local_Permission[This.CmdContainer, CmdContainer.t0.y] > 0.0;
    assert Heap[This.CmdContainer, CmdContainer.t0.x] == Heap[
        This.CmdContainer, CmdContainer.t0.y];
    //Remove Permisssion
    local_Permission[This.CmdContainer, CmdContainer.t0.x] := 0.0;
    local_Permission[This.CmdContainer, CmdContainer.t0.y] := 0.0;
}
```

Boogie Translation

4.2.9.9 Call Command

Translation of call command first copy the heap into an old heap, and check involving expression are defined. To establish a call, it is the caller's responsibility to check the pre-condition and provide the necessary permission required in the method *takes* clause and loss the same amount of permission. After the method call, callee receives the amount of permission(s) mentioned in *gives* clause, assert the post-condition. Pre-condition should also assume that the untouched parts of the heap have not changed. New variables are declared to copy the input and output arguments. After the callee thread received permission from the method, a new variable *that* is also declared to copy the current object reference.

```
(class Counter()
proc inc(in x: Int32)
  takes count;
  pre count > 0 && x >= 0
  post count' = count + x;
  gives count;
  obj count : Int32 := 0;
(thread(*t0*)
  (while true
    do
      (accept inc(int x: Int32)
        count := count + x;
        accept)
      thread)
  (thread (*t1*) claim count;
    inc(20);
    assert count = 20;
  thread)
class)
```

```

const unique Counter: ClassName;
const unique Counter.count: Field int;
const unique Counter.inc.x: Field int;
const unique Counter.i: Field int;
const unique Counter.iCopy: Field int;
procedure counter.constructor(This.Counter: Ref)
  requires dtype(This.Counter) <: Counter;
  modifies Heap;
{
  var oldHeap: HeapType;
  var Permission, oldPermission: PermissionType;
  Heap[This.Counter, Counter.count] := 0;
  Permission[This.Counter, Counter.count] := 1.0;
  assert isInt32(Heap[This.Counter, Counter.count]);
  assert (forall<x> r: Ref, f : Field x :: Permission[r, f]
    <= 1.0);
  havoc oldHeap;
  oldHeap := Heap;
}
procedure counter.t0(This.Counter: Ref)
  requires dtype(This.Counter) <: Counter;
  modifies Heap;
{
  var oldPermission, Permission : PermissionType;
  var oldHeap: HeapType;
  while(true)
    invariant (forall<x> r: Ref, f: Field x :: !(r == This.
      Counter && f == Counter.count) ==> Heap[r, f] ==
      oldHeap[r, f]);
  {
    goto inc;
    inc:
      // Takes Permission
      assert Permission[This.Counter, Counter.count] == 1.0;
      Permission[This.Counter, Counter.count] := Permission[This.
        Counter, Counter.count] - 1.0;

      // Assume Pre Condition
      assert Permission[This.Counter, Counter.count] > 0.0;
      assume Heap[This.Counter, Counter.count] > 0 && Heap[This.
        Counter, Counter.inc.x] >= 0;

      //Body
      assert Permission[This.Counter, Counter.count] == 1.0 &&

```

```

    Permission [ This . Counter , Counter . inc . x ] > 0.0;
    Heap [ This . Counter , Counter . count ] := Heap [ This . Counter ,
        Counter . count ] + Heap [ This . Counter , Counter . inc . x ];

    //Assert Post Condition
    assert Permission [ This . Counter , Counter . count ] > 0.0 &&
        Permission [ This . Counter , Counter . inc . x ] > 0.0;
    assert Heap [ This . Counter , Counter . count ] == (oldHeap [ This .
        Counter , Counter . count ] + Heap [ This . Counter , Counter .
        inc . x ]);

    //Gives Permission
    Permission [ This . Counter , Counter . count ] := Permission [ This
        . Counter , Counter . count ] + 1.0 ;
    goto end;
    end:
}
}

procedure counter . t1 (This . Counter : Ref)
requires dtype (This . Counter ) <: Counter ;
modifies Heap ;
{
    var that : Ref;
    var oldHeap : HeapType;
    var Permission : PermissionType;
    oldHeap := Heap;
    that := This . Counter;

    //Copy Arguments
    assert isInt32 (Heap [ This . Counter , Counter . i ]);
    Heap [ This . Counter , Counter . iCopy ] := Heap [ This . Counter ,
        Counter . i ];
    assert isInt32 (Heap [ This . Counter , Counter . iCopy ] );

    // Assert Pre Condition
    assert Heap [ that , Counter . count ] >= 0 && Heap [ This . Counter ,
        Counter . iCopy ] > 0;

    //Takes Permission : Remove
    assert Permission [ This . Counter , Counter . count ] == 1.0;
    Permission [ This . Counter , Counter . count ] := Permission [ This .
        Counter , Counter . count ] - 1.0;

    //Add Permission
    Permission [ that , Counter . count ] := Permission [ that , Counter .

```

```

    count] + 1.0;

// Post Condition
oldHeap := Heap;
havoc Heap;
assume (Heap[that, Counter.count] == oldHeap[that, Counter.
    count] + Heap[This.Counter, Counter.iCopy]);
assert Heap[This.Counter, Counter.count] == 20;
}

```

Boogie Translation

4.2.9.10 With Command

The *with* command's translation is based on the locks which infer the class invariant. Translation of lock is described in the following steps:

- Assume class invariant at the entrance of the lock
- Declare variable *that* for current object reference which replace the *this* translation in lock.
- Assume the permission specifications in class invariant, permission not specified in invariant are assumed to be zero
- Translate the conditional part of invariant, assert the conditional part's definedness, and assume that part.
- If the lock is annotated with *takes* specification, the mentioned permissions in *takes* class is transferred to thread.
- Statements inside the lock are translated where permission of lock (*lock-Permission*) and thread permission are combined to check the locations are readable and writable.

- Permissions are removed from thread mentioned under *gives* clause.

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

HARPO Code

```
/*Boogie prelude*/
const unique Counter:ClassName;
const unique Counter.count:Field int;
procedure Counter.constructor(This_Counter:Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
{
  var Permission : PermissionType where (forall <x> r: Ref, f:
    Field x :: Permission[r,f] == 0.0);
  var oldHeap:HeapType;
  havoc Heap;
  //Claim
  Permission[This_Counter,Counter.count] := Permission[
    This_Counter,Counter.count] + 0.5;
  assert (forall<x> r: Ref, f: Field x :: Permission[r,f] <=
    1.0);
```

```

// Initialize Heap
Heap[This_Counter, Counter.count] := 0;
// Class Invariant
assert Permission[This_Counter, Counter.count] > 0.0 && Heap
    [This_Counter, Counter.count] >= 0;
}
procedure Counter.t#0 (This_Counter : Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
modifies LockPermission;
{
var Permission: PermissionType where (forall<x> r: Ref, f:
    Field x :: Permission[r, f] == 0.0);
var oldPermission: PermissionType where (forall<x> r: Ref, f:
    : Field x :: oldPermission[r, f] == 0.0);
var oldHeap, preHeap, tempHeap: HeapType;
var this_lock: Ref;
while( true)
    invariant (forall<x> r: Ref, f: Field x :: 0.0 <=
        Permission[r, f] && Permission[r, f] <= 1.0);
    {
        goto increment;
        increment:
        //Taking Permission(s)
        oldPermission := Permission;
        if(Permission[This_Counter, Counter.count] == 0.0)
        {
            havoc tempHeap;
            Heap[This_Counter, Counter.count] := tempHeap[
                This_Counter, Counter.count];
            Permission[This_Counter, Counter.count] := Permission
                [This_Counter, Counter.count]+0.5;
        }
        //Pre Condition(s)
        oldHeap := Heap;
        havoc Heap;
        assume Heap[This_Counter, Counter.count] >= 0;
        //Method Implementation
        this_lock:=This_Counter;
        preHeap := Heap;
        havoc tempHeap;
        havoc LockPermission;
        assume LockPermission[this_lock, Counter.count] > 0.0 &&
            Heap[This_Counter, Counter.count] >= 0;
        //Check Assignment Definedness
    }
}

```

```

assert LockPermission [ This_Counter , Counter . count ] +
    Permission [ This_Counter , Counter . count ] == 1.0 ;
assert LockPermission [ This_Counter , Counter . count ] > 0.0;
assert Permission [ This_Counter , Counter . count ] > 0.0;
//Check Assignment Definedness Ends
//Assignment Command
Heap [ This_Counter , Counter . count ] := Heap [ This_Counter ,
    Counter . count ] + 1;
//Assignment Command Ends
//Assert definedness of object invariant and assume object
    invariance
assert LockPermission [ this_lock , Counter . count ] > 0.0;
assert LockPermission [ this_lock , Counter . count ] > 0.0 &&
    Heap [ This_Counter , Counter . count ] >= 0;
//Post Condition(s)
assert Permission [ This_Counter , Counter . count ] > 0.0;
assert Heap [ This_Counter , Counter . count ] > 0;
//Giving Permissions(s)
assert Permission [ This_Counter , Counter . count ] >= 0.5;
Permission [ This_Counter , Counter . count ] := Permission [
    This_Counter , Counter . count ] - 0.5;
goto end;
end:
}
}//end of Thread Procedure

```

Boogie Translation

4.2.9.11 Co Command Parallelism

The ‘co’ blocks give a way to parallelize the parts of a program. A ‘co’ block can have a separate claim on location(s). While translation, thread losses permission on all the claimed locations in the ‘co’ block.

New variables *Permission-1*,*Permission-2*,...*Permission-n* are declared respectively for each ‘co’ command. Then the statements of each child thread are translated under respective permission.

```

(class Math()
procedure Sum()
  takes x
  pre x >= 0;
  post x' = x+2;
  gives x;
obj x:Int32 := 0;
(thread (* t1*)
(while true
  do
    (accept Sum()
      obj sum0 : Int32 := 0;
      obj sum1 : Int32 := 0;
      (co
        claim x@0.5 , sum0@1.0
        sum1 := x + 1;
      ||
        claim x@0.3 , sum1@1.0
        sum1 := x + 1;
      co)
    accept)
  while)
thread)
class)

```

HARPO Code

```

/*Boogie prelude*/
const unique Math: ClassName;
const unique Math.x : Field int;
const unique Math.t1.sum0 : Field int;
const unique Math.t1.sum1 : Field int;
procedure Math.constructor(This.Math: Ref)
  requires dtype(This.Math)<: Math;
  modifies Heap;
{
  var Permission: PermissionType;
  Heap[This.Math, Math.x] := 0;
  Heap[This.Math, Math.t1.sum0] := 0;
  Heap[This.Math, Math.t1.sum1] := 0;
  Permission[This.Math, Math.t1.sum0] := 1.0;
  Permission[This.Math, Math.t1.sum1] := 1.0;
  Permission[This.Math, Math.x] := 0.5;
}

```

```

Permission [ This . Math , Math . x ] := 0 . 3 ;
// assert ( forall < x > r : Ref , f : Field x :: r == This . Math , f
    == Math . x ==> Permission [ r , f ] <= 1 . 0 ) ;
}

procedure Math . t1 ( This . Math : Ref )
    requires dtype ( This . Math ) <: Math ;
    modifies Heap ;
{
    var oldHeap : HeapType ;
    var Permission : PermissionType ;
    var Permission_1 , Permission_2 : PermissionType ;
    assert Permission [ This . Math , Math . x ] >= 0 . 5 ;
    Permission [ This . Math , Math . x ] := Permission [ This . Math , Math
        . x ] + 0 . 5 ;
    Permission [ This . Math , Math . t1 . sum0 ] := Permission [ This . Math
        , Math . t1 . sum0 ] - 1 . 0 ;
    assert Permission [ This . Math , Math . x ] >= 0 . 25 ;
    Permission [ This . Math , Math . x ] := Permission [ This . Math , Math
        . x ] - 0 . 25 ;
    assert Permission [ This . Math , Math . t1 . sum1 ] == 1 . 0 ;
    Permission [ This . Math , Math . t1 . sum1 ] := Permission [ This . Math ,
        Math . t1 . sum1 ] - 1 . 0 ;

//Co Block 1
Permission_1 [ This . Math , Math . x ] := Permission_1 [ This . Math ,
    Math . x ] + 0 . 5 ;
Permission_1 [ This . Math , Math . t1 . sum0 ] := Permission_1 [ This .
    Math , Math . t1 . sum0 ] + 1 . 0 ;

//block 1 translation

//Co Block 2

Permission_2 [ This . Math , Math . x ] := Permission_2 [ This . Math ,
    Math . x ] + 0 . 25 ;

Permission_2 [ This . Math , Math . t1 . sum1 ] := Permission_2 [ This .
    Math , Math . t1 . sum1 ] + 1 . 0 ;

//block 2 translation

//Add Permission Back
}

```

Chapter 5

HARPO Verifier Design

The HARPO verifier is implemented in an object-oriented fashion. To understand the design of the HARPO verifier, it is necessary first to understand how we intend to verify a HARPO program. This chapter contains a detailed design of the HARPO verifier.

5.1 Compiler Overview

The HARPO compiler entails standard passes of compilation and three different backends. Figure B.1 in the appendix depicts the package diagram of the HARPO compiler. Figure 5.1 depicts the stages of the compilation process into other source programs.

- HARPO Source: is a standard program written in HARPO Language

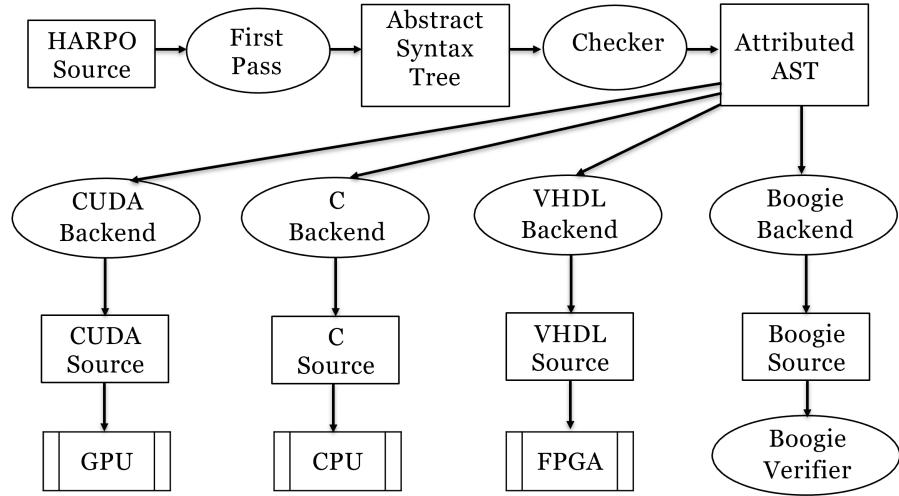


Figure 5.1: HARPO Compiler Implementation Overview

- First Pass: is a parsing stage where parser parse the program text by passing it through lexical and syntax analyzers and produces an Abstract Syntax Tree (AST).
- Abstract Syntax Tree: AST is the output from the first pass.
- Checker Pass: both add attributes to the AST by checking the names with their declarations, creating symbol tables, and linking them to their declarations (i.e., resolution), type creation, type conversion, and, class environment creation.
- Attributed Abstract Syntax Tree: an attributed AST is the final output from the checking pass.
- Boogie backend: traverse the AST and translate it into Boogie code.
- Boogie Code: is the final output from Boogie backend.
- C Backend: traverse the AST and generate C code.

- C Code: is the final output from C backend.
- VHDL backend: traverse the attributed AST and generate the equivalent VHDL code.
- VHDL Code: final output from VHDL backend.

5.2 HARPO Verifier Overview

Chapter 2 contains the design of annotations used in verification. These annotations are the part of HARPO syntax. These annotations are not required to generate the C or VHDL code from HARPO, but for generating the Boogie code, these annotations are mandatory. Figure 5.2 depicts the verification data flow explaining the process of verification.

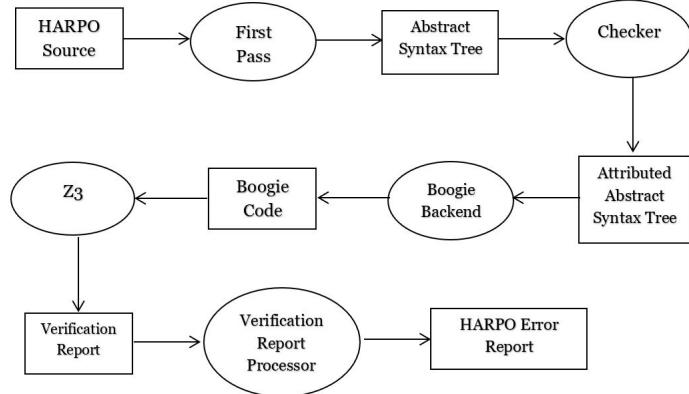


Figure 5.2: Translation Data Flow

The front-end of the HARPO compiler is the same for all types of code generation phases.

5.2.1 Front-end Compilation

The front-end contains the parsing and checking phases. In parsing, complete HARPO program with annotations is parsed, and in the checking phase, the AST is checked and updated. Annotations converted into the AST are entirely ignored by the other backends and used only in the Boogie backend.

- Parsing: we have used a recursive descent parser generated with JavaCC.

The parser analyzes the HARPO character stream and converts it into a token stream used by a syntax analyzer to create syntax trees. Parser makes calls to the builder and creates an AST. Builder is coded in *Scala*, and it uses *AST* package containing case classes for generating abstract syntax tree. Consider a simple class written in HARPO below:

```
(class Bob()  
  
  ghost obj c: Int32 := 0  
  
  class)
```

has a class declaration named *Bob* containing a ghost object declaration name *c*. Parser will generate the abstract syntax tree show in Figure 5.3.

- Checker: takes abstract syntax tree generated by the parser and adds attributes into it as shown in Figure 5.4

5.2.2 Abstract Syntax Tree Components.

Abstract Syntax Tree (AST) generated with various recognized nodes in the front-end. The collection of these nodes represents the program as AST. Top

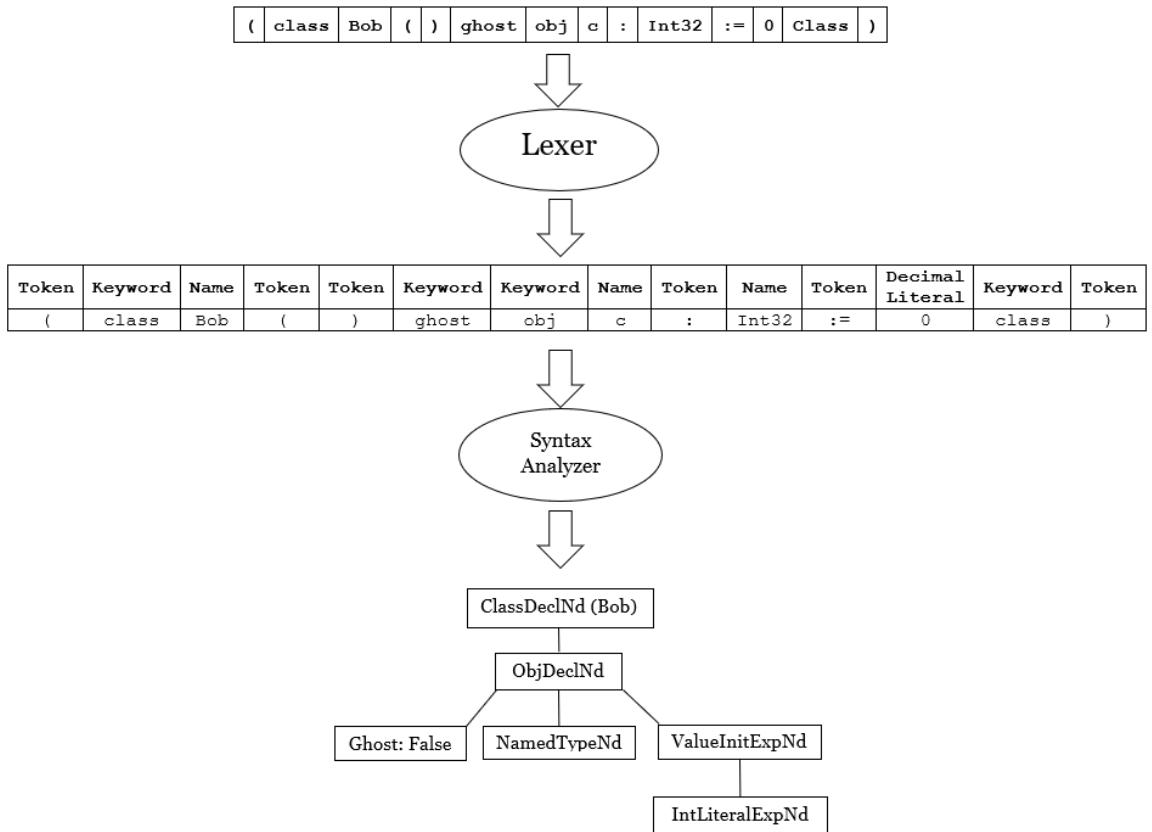


Figure 5.3: Parsing character stream into token stream and generating the AST

level nodes are presented in simplified UML diagram shown Figure 5.5.

5.2.3 Checker Passes

The Checker runs multiple passes and makes a AST passed on to Boogie backend. Figure 5.6 depicts checking passes.

- Symbol Table Maker: creates a map from declarations to their fully qualified names.

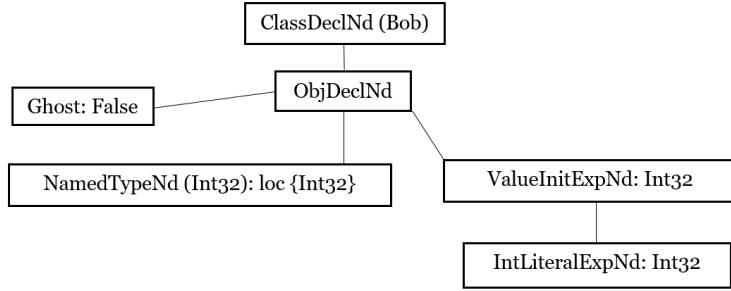


Figure 5.4: Attributed AST

- Resolver: links each name node to its declaration, where the symbol table is used to resolve the names after resolution.
- Type Creation: creates the required types and associates all type nodes to their respective concrete types.
- Class Environment Creator: creates a representation of the class, which makes it easy to find fields and methods inside the class.
- Type Checker: checks for the suitable types to associate Expression Nodes, Initialization Expression Nodes, and No-Type Nodes. Also, inserts fetch nodes while value conversion and check the correctness of assigned types.

5.2.4 Boogie back-end

Boogie backend takes the AST and traverses all the declarations nodes inside-out to generate Boogie source code. Boogie code has two parts: static prelude and a customized code generated from attributed AST. Figure 5.7 depicts the data flow of Boogie backend.

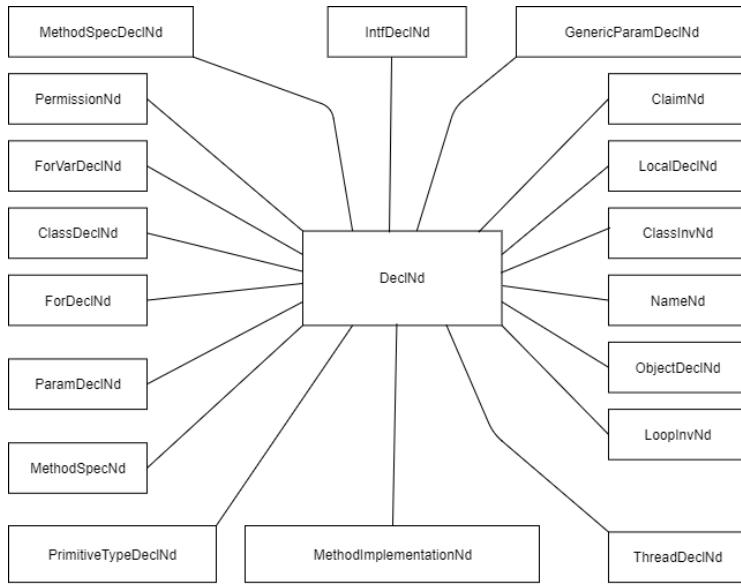


Figure 5.5: Top level UML Diagram containing parent and child nodes inside AST Package.

Prelude is independent of the program being translated. Prelude contains some important properties that need to be defined before the actual translation comes: memory modeling, reference types, type axioms, permission types, and array length. Prelude has a heap memory model that maps the fields and object references to values.

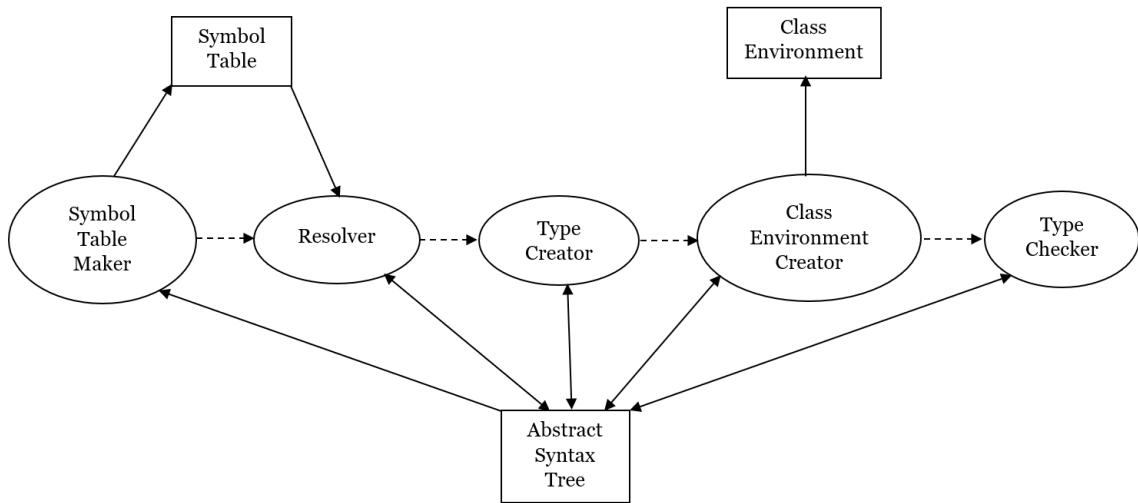


Figure 5.6: Checker Phase

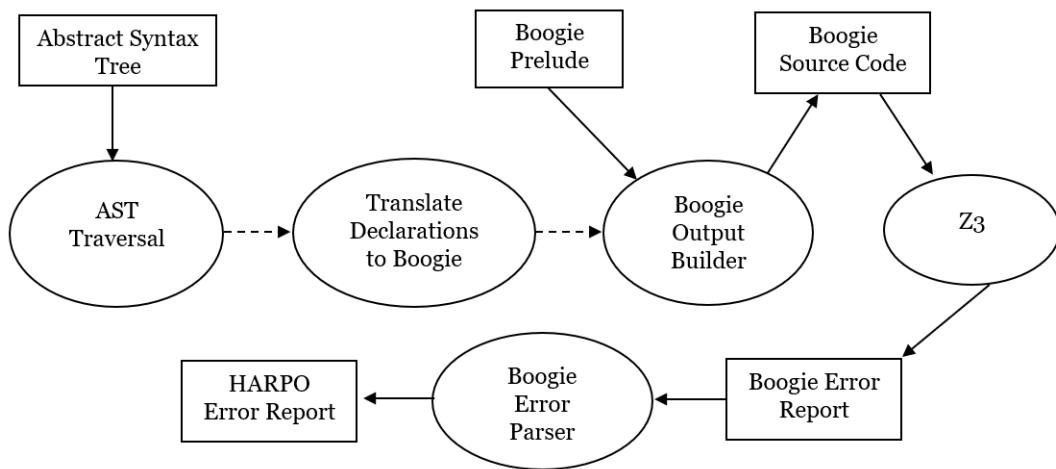


Figure 5.7: Complete Boogie Backend

Chapter 6

Automated Verification

Automated translation takes annotated HARPO programs as input and produces Boogie code, which generates the verification conditions validated by *Z3*. This chapter contains an example HARPO program, and each verification stage of the example program is described in detail. In Section 6.5, some unit tests generating the Boogie code are reported. Figure 6.1 shows the data flow for generating the Boogie code from checked AST.

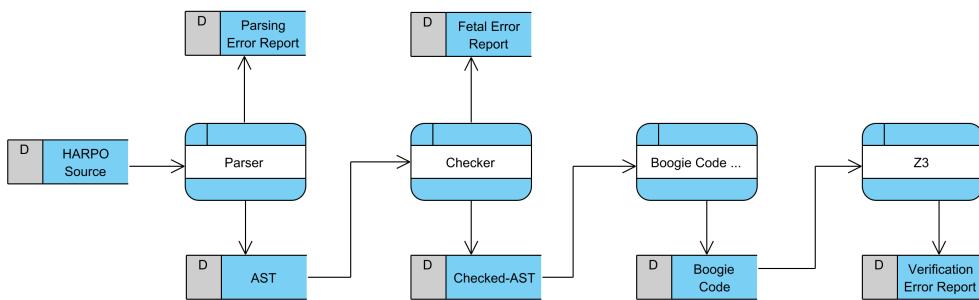


Figure 6.1: Verification Data Flow

6.1 HARPO Source Code

Consider a HARPO program in Listing 6.1.1 given below. Counter class claims half permission ‘0.5’ on *count* field. Class *invariant* says read permission on *count* must remain valid, and value of *count* field must be greater than or equal to ‘0’. A public procedure named *increment* takes half permission ‘0.5’ from thread calling this procedure and check to see the *count* value must be greater or equal to ‘0’ before the call is made. After the procedure’s execution, it gives back the amount of permission to calling thread and asserts the count value is positive. Thread **t0** contains a while loop always accepting the *increment* procedure implementation. The *with* command inside procedure implements lock. The currently locked object infers the permissions from class invariant, and with the presence of a lock, lock permission and thread permission added to make an assignment. Thread **t0** does not have permission on the *count* field, and so it is not possible to make this assignment resulting in a verification error. This verification error should be translated back to a meaningful HARPO error message, which points the line number of *counter* class with a message.

```
1 (class Counter()
2   claim count@0..5
3   invariant canRead(count) /\ count >_ 0
4   public proc increment()
5     takes count@0..5
6     pre count >_ 0
7     post count' > 0
8     gives count@0..5
9     obj count: Int32 := 0
10    (thread (*t0*)
11      (while true
12        do
13          (accept increment())
```

```

14     (with this
15       do
16         count := count+1
17       with)
18     accept)
19   while)
20   thread)
21 class)

```

Listing 6.1.1: The ‘*Counter*’ class example used in automated translation

6.2 Translation

Translation of the Listing 6.1.1 involves three significant steps. All three are explained in the subsequent sections. These steps include generation, manipulation, and use of AST. A generated AST contains several declarations and expression nodes. For instance Figure 6.2 shows a simplified AST representation of *Counter* class given below.

6.2.1 Parser

HARPO parser takes the source code of counter class and converts it into a legitimate abstract syntax tree. If there is any syntactical error in the source code, it will be pointed out by parser at this stage. Given below is the auto-generated and then simplified AST of *Counter* class.

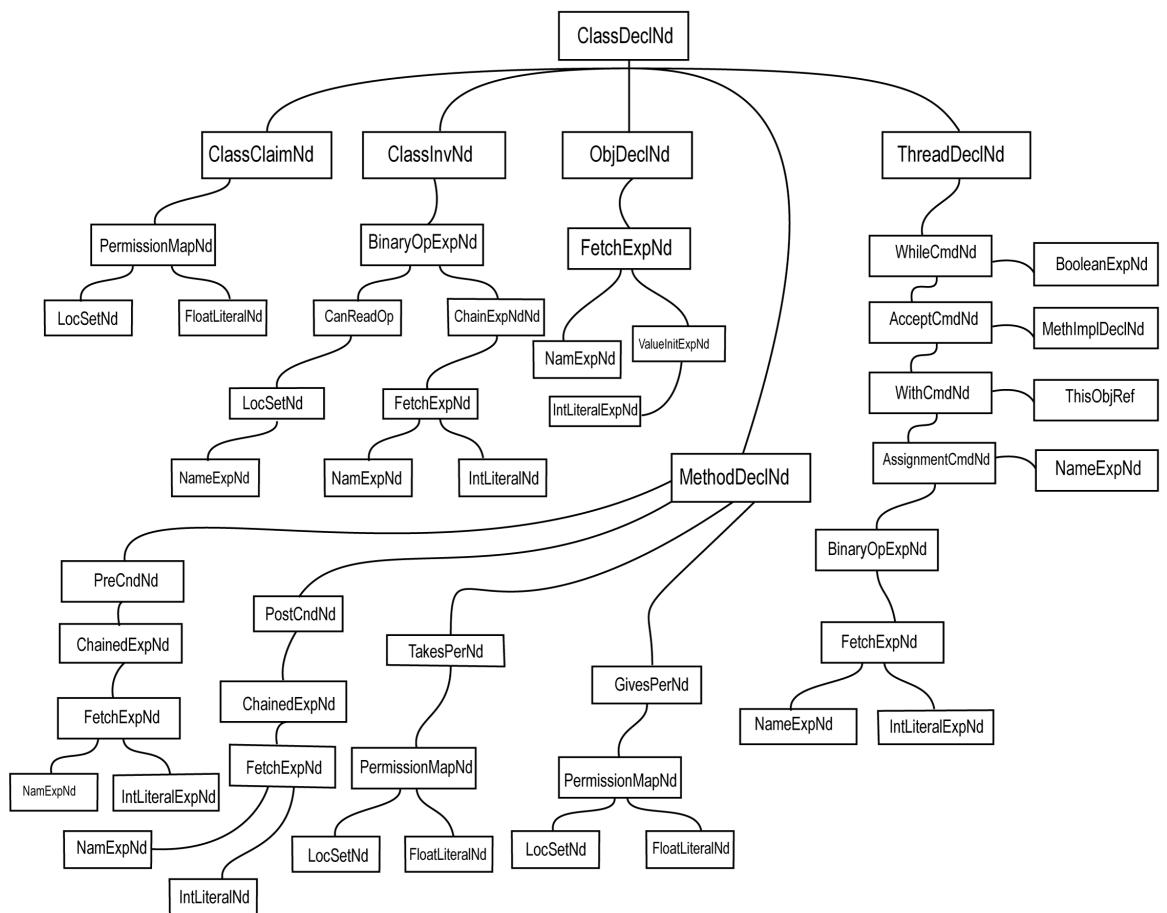


Figure 6.2: Abstract Syntax Tree of *Counter* Class

```
[ ClassDeclNd(
  [ ClaimNd[ClaimNd#0](
    PermissionMapNd[PermMapNd#1](
      [ LocSetNd( NameExpNd( count ) : NONE ) ],
      [ FloaLiteralExpNd( 0.5 ) : NONE ] ) ),
  ClassInvNd[*inv*0](
    BinaryOpExpNd(
      AndOp,
      CanReadOp( LocSetNd( NameExpNd( count ) : NONE
    ) ) : NONE,
      ChainExpNd(
        [ GreaterOrEqualOp ],
        [ NameExpNd( count ) : NONE, IntLiteralExpNd
( 0 ) : NONE ] )
      : NONE )
```



```

        : NONE ] ) ),
SkipCmdNd( ) )
: NONE ] ) ) ] ) ]

```

Listing 6.2.1: Internal Representation of Simplified Abstract Syntax Tree Generated by Parser

6.2.2 Checker

The checker pass takes the tree shown in Listing 6.2.1 and creates a symbol table for each declaration in the tree. The checker resolves all the names used in the tree and links them with their appropriate declaration. For instance, *count* name used in assignment command linked with *count* field in their resolution phase. If the name is not as defined in the declaration, then the resolution pass will report an error. After resolution, standard declarations are added, all type names are checked, fields are promoted to locations, and after type creation, all expressions and their types are checked for required type conversions. Listing 6.2.2 shows the attributed AST after the checking phase.

```

[ 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 )
        : NONE ),
MethodDeclNd(PublicAccess)
[ PreCndNd(
    ChainExpNd(
        [ GreaterOrEqualOp ],
        [ FetchExpNd( NameExpNd( count ) : loc{Int32}
} ) : Int32 ,
        IntLiteralExpNd( 0 ) : Int32 ] )
        : Bool ) ]
[ PostCndNd(
    ChainExpNd(
        [ GreaterOp ],
        [ UnaryOpExpNd(
            PrimeOp ,
            FetchExpNd( NameExpNd( count ) : loc{
Int32} ) : Int32 )
        : Int32 ,
        IntLiteralExpNd( 0 ) : Int32 ] )
        : Bool ) ]
[ GivesPerNd(
    PermissionMapNd[PermMapNd#3](
        [ LocSetNd( NameExpNd( count ) : loc{Int32}
) ], 
        [ FloatLiteralExpNd( 0.5 ) : Real64 ] ) ) ]
[ TakesPerNd(
    PermissionMapNd[PermMapNd#2](
        [ LocSetNd( NameExpNd( count ) : loc{Int32}
) ], 
        [ FloatLiteralExpNd( 0.5 ) : Real64 ] ) ) ]
: method List(),
ObjDeclNd[count](
    Ghost :false,
    NamedTypeNd( Int32 ) : loc{Int32},
    ValueInitExpNd( IntLiteralExpNd( 0 ) : Int32 ) :
Int32 ),
ThreadDeclNd[t#0](
    WhileCmdNd(
        BooleanLiteralExpNd( true ) : Bool,
        AcceptCmdNd(
            [ MethodImplementationDeclNd(
                increment ,
                NoExpNd( ) : NONE ,
                WithCmdNd(
                    ThisObjRef( this ) : NONE ,

```

```

        NoExpNd( ) : NONE ,
AssignmentCmdNd(
    [ NameExpNd( count ) : loc{Int32}
],
    [ BinaryOpExpNd(
        AddOp ,
        FetchExpNd( NameExpNd( count ) :
loc{Int32} ) : Int32 ,
            IntLiteralExpNd( 1 ) : Int32
            : Int32 ] ) ),
    SkipCmdNd( ) )
: method List() ] ) ) ] ),
PrimitiveTypeDeclNd( Bool ),
PrimitiveTypeDeclNd( Int8 ),
PrimitiveTypeDeclNd( Int16 ),
PrimitiveTypeDeclNd( Int32 ),
PrimitiveTypeDeclNd( Int64 ),
PrimitiveTypeDeclNd( Real16 ),
PrimitiveTypeDeclNd( Real32 ),
PrimitiveTypeDeclNd( Real64 ) ]

```

Listing 6.2.2: Abstract Syntax Tree after Checker Pass

6.2.3 Code Generator

Boogie code generator takes the AST updated in checker passes and generates the Boogie code. The code generator uses a string builder to merge the output strings into a single large string format. The Code generator has recursive calls to subroutines, i.e., type code generation, expression code generation, and command code generation.

6.3 Boogie Source Code

After the code generation phase, we get the desired Boogie code translated from the HARPO code. Then, we pass the Boogie source code to the Boogie verifier for generating the verification conditions. Boogie verifier generates verification conditions, checks them with $Z3$, and returns the verification report. Listing 6.3.1 is the Boogie code generated from *Counter* class.

```
1 // Prelude begin
2 type Ref;
3 type Field x;
4 type HeapType = <x> [Ref,Field x]x;
5 var Heap:HeapType;
6 type ArrayRef x;
7 type ArrayHeapType = <x> [ArrayRef x, int]x;
8 var ArrayHeap:ArrayHeapType;
9 type Perm = real;
10 type PermissionType = <x> [Ref,Field x]Perm;
11 type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
12 var LockPermission: PermissionType;
13 var ArrayLockPermission: ArrayPermissionType;
14 function Length<x>(Field (ArrayRef x)) returns (int);
15 const unique min8:int;
16 axiom min8 == -128;
17 const unique max8:int;
18 axiom max8==127;
19 const unique min16:int;
20 axiom min16 == -32768;
21 const unique max16:int;
22 axiom max16 == 32767;
23 const unique min32:int;
24 axiom min32 == -2147483648;
25 const unique max32:int;
26 axiom max32 == 2147483647;
27 const unique min64:int;
28 axiom min64 == -9223372036854775808;
29 const unique max64:int;
30 axiom max64== 9223372036854775807;
31 function Isint8(int) returns (bool);
32 axiom (forall x:int :: Isint8(x) <=> min8 <=x&&x<=max8);
33 function Isint16(int) returns (bool);
```

```

34 axiom (forall x:int :: Isint16(x) <==> min16 <=x&&x<=max16
35 );
36 function Isint32(int) returns (bool);
37 axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
38 );
39 function Isint64(int) returns (bool);
40 axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
41 max64);
42 const unique minPer : Perm;
43 axiom minPer == 0.0;
44 const unique maxPer : Perm;
45 axiom maxPer == 1.0;
46 function IsValidPermission(Perm) returns (bool);
47 axiom (forall x: Perm:: IsValidPermission(x) <==> minPer
48 <= x && x <= maxPer);

49 const unique TenPow16 : real;
50 axiom TenPow16 == 1000000000000000.0; // 10 ^ 16
51 const unique TenPow32 : real;
52 axiom TenPow32 == 10000000000000000000000000000000.0; //
53 10 ^ 32
54 const unique TenPow64 : real;
55 axiom TenPow64 ==
56 100000000000000000000000000000000000000000000000000000000000000
57 000000000000.0; // 10 ^ 64

58
59 const unique minReal16:real;
60 axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
61 const unique maxReal16:real;
62 axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

63
64 const unique minReal32:real;
65 axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
66 3.402823466 E - 38
67 const unique maxReal32:real;
68 axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
69 //3.402823465 E + 38

70
71 const unique minReal64:real;
72 axiom minReal64 == -1.7976931348623157 / (TenPow64*
73 TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0);
74 // -1.7976931348623157 E - 308
75 const unique maxReal64:real;
76 axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64
77 *TenPow64*TenPow64*TenPow32*TenPow16*1000.0); //

```

```

1.7976931348623156 E + 308

68
69 function isReal16(real) returns (bool);
70 axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
71   x <= maxReal16);
72
73 function isReal32(real) returns (bool);
74 axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
75   x <= maxReal32);
76
77 function isReal64(real) returns (bool);
78 axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
79   x <= maxReal64);
80
81
82 type ClassName;
83 function dtype(Ref) returns (ClassName);
84 // Prelude ends
85 const unique Counter:ClassName;
86 const unique Counter.count:Field int;
87 procedure Counter.constructor(This_Counter:Ref)
88 requires dtype(This_Counter) <: Counter;
89 modifies Heap;
90 {
91   var Permission : PermissionType where (forall <x> r: Ref,
92     f: Field x :: Permission[r,f] == 0.0);
93   var oldHeap:HeapType;
94   havoc Heap;
95   //Claim
96   Permission[This_Counter,Counter.count] := Permission[
97     This_Counter,Counter.count] + 0.5;
98   assert (forall <x> r: Ref, f: Field x :: Permission[r,f]
99     <= 1.0);
100  //Initialize Heap
101  Heap[This_Counter,Counter.count] := 0;
102  //Class Invariant
103  assert Permission[This_Counter,Counter.count] > 0.0 &&
      Heap[This_Counter,Counter.count] >= 0;
}
procedure Counter.t#0 (This_Counter : Ref)
  requires dtype(This_Counter) <: Counter;

```

```

104    modifies Heap;
105    modifies LockPermission;
106    {
107        var Permission: PermissionType where (forall<x> r: Ref, f
108            : Field x :: Permission[r,f] == 0.0);
109        var oldPermission: PermissionType where (forall<x> r: Ref
110            , f: Field x :: oldPermission[r,f] == 0.0);
111        var oldHeap, preHeap, tempHeap: HeapType;
112        var this_lock: Ref;
113        while( true)
114            invariant (forall<x> r: Ref, f: Field x :: 0.0 <=
115                Permission[r,f] && Permission[r,f] <= 1.0);
116            {
117                goto increment;
118                increment:
119                    //Taking Permission(s)
120                    oldPermission := Permission;
121                    if(Permission[This_Counter,Counter.count] == 0.0)
122                    {
123                        havoc tempHeap;
124                        Heap[This_Counter,Counter.count] := tempHeap[
125                            This_Counter ,Counter.count];
126                        Permission[This_Counter,Counter.count] := Permission
127                            [This_Counter,Counter.count]+0.5;
128                    }
129                    //Pre Condition(s)
130                    oldHeap := Heap;
131                    havoc Heap;
132                    assume Heap[This_Counter,Counter.count] >= 0;
133                    //Method Implementation
134                    this_lock:=This_Counter;
135                    preHeap := Heap;
136                    havoc tempHeap;
137                    havoc LockPermission;
138                    assume LockPermission[this_lock,Counter.count] > 0.0
139                    && Heap[This_Counter,Counter.count] >= 0;
140                    //Check Assignment Definedness
141                    assert LockPermission[This_Counter,Counter.count] +
142                        Permission[This_Counter,Counter.count] == 1.0 ;
143                    assert LockPermission[This_Counter,Counter.count] >
144                        0.0;
145                    assert Permission[This_Counter,Counter.count] > 0.0;
146                    //Check Assignment Definedness Ends
147                    //Assignment Command

```

```

140     Heap[This_Counter,Counter.count] := Heap[This_Counter,
141         Counter.count] + 1;
142     //Assignment Command Ends
143     //Assert definedness of object invariant and assume
144     //object invariance
145     assert LockPermission[this_lock,Counter.count] > 0.0;
146     assert LockPermission[this_lock,Counter.count] > 0.0
147         && Heap[This_Counter,Counter.count] >= 0;
148     //Post Condition(s)
149     assert Permission[This_Counter,Counter.count] > 0.0;
150     assert Heap[This_Counter,Counter.count] > 0;
151     //Giving Permissions(s)
152     assert Permission[This_Counter,Counter.count] >= 0.5;
153     Permission[This_Counter,Counter.count] := Permission[
154         This_Counter,Counter.count] - 0.5;
155     goto end;
156 end:
157 }
158 } //end of Thread Procedure

```

Listing 6.3.1: Boogie Code Generated from 6.2.2

6.4 Boogie Error Report

Boogie verifier inspects the code by generating the verification conditions and passing them to *Z3*, which returns the verification report either with a list of errors and their traces, or in case of no errors, it reports successful verification. Execution traces help reach the root cause of certain verification error(s) unless there are potentially multiple root causes, and we find one of them. Boogie verifier reports the following errors for Listing 6.3.1 verification results.

```

1  input(98,4): Error BP5001: This assertion might not hold.
2  Execution trace:
3      input(74,2): anon0
4      input(74,2): anon4_LoopHead
5      input(77,4): anon4_LoopBody
6      input(83,6): anon5_Then

```

```

7      input(88,12): anon3
8 Boogie program verifier finished with 1 verified, 1 error

```

Listing 6.4.1: Boogie Verifier Output from Listing 6.3.1

```

1 Error report from HARPO verifier
2 Fatal errors = 0
3 Warning(s) = 0
4 Verification errors = 1
5 File: HarpoSourceCode.harpo line: 3 column: 29 Does not
   have sufficient permission to do assignment.

```

Listing 6.4.2: HARPO Verifier Output from Listing 6.1.1

By replacing line 3 of Listing 6.1.1 to following class invariant:

```

.
.
invariant permission(count) = 0.5 /\ count >= 0
.
.
```

Listing 6.4.3: Making Permission to do Write in Listing 6.1.1

code generator will generate the assumption and assertion shown in Listing 6.4.4

which was root cause of error in shown in 6.4.1.

```

assume LockPermission[this_lock,Counter.count] == 0.5
  && Heap[This_Counter,Counter.count] >= 0;
//Check Assignment Definedness
assert LockPermission[This_Counter,Counter.count] +
  Permission[This_Counter,Counter.count] == 1.0 ;

```

Listing 6.4.4: Assuming Write Permissions after 6.4.3

The output of the HARPO verifier will be as follows:

```

Error report from HARPO verifier
Fatal errors = 0
Warning(s) = 0
Verification errors = 0
Successfully Verified with No Verification Errors.

```

Listing 6.4.5: HARPO Verifier Output

Appendix A contains detailed case studies on HARPO to Boogie automated translation.

6.5 Code Generator Tests

Code generation tests are applied to test the HARPO verifier code generation phase. These tests show Boogie code generation of various HARPO program components. Each declaration, command, and expression is tested independently. For each HARPO code example in these tests, regular expressions are applied to retrieve required Boogie strings from the large output generated code. Following each sub-section contains some input HARPO code, applied regular expression, and Boogie code extracted from large Boogie string are provided. The remaining code generation tests are provided in A.1.

6.5.1 Class Declaration

```
(class Math()  
// Class Code  
class)
```

Listing 6.5.1: Class Declaration

```
".*const *unique *Math *: *ClassName *; .*"
```

Listing 6.5.2: Regular Expression

```
const unique Math:ClassName;
```

Listing 6.5.3: Boogie Code Matched for Listing 6.5.1

6.5.2 Interface Declaration

```
(interface MyInterface  
// Interface Body  
interface)
```

Listing 6.5.4: Interface Declaration

```
".*const +unique +MyInterface *: *ClassName *;.*"
```

Listing 6.5.5: Regular Expression

```
const unique MyInterface : ClassName;
```

Listing 6.5.6: Boogie Code Matched for Listing 6.5.4

6.5.3 Object Declaration

```
(class Math()  
  obj c: Int32:=0;  
class)
```

Listing 6.5.7: Object Declaration

```
".*const *unique *Math.c *: *Field *int *;.*"  
".*Heap\\[This.Math,Math.c\\] *:= *0 *;.*"
```

Listing 6.5.8: Regular Expression

```
const unique Math.c:Field int;  
Heap [This.Math,Math.c] := 0 ;
```

Listing 6.5.9: Boogie Code Matched for Listing Listing 6.5.7

6.5.4 Ghost Object Declaration

```
(class Math()  
  ghost obj c: Int32:=0;  
class)
```

Listing 6.5.10: Ghost Object Declaration

```
".*const *unique *Math.c *: *Field *int *;.*"  
".*Heap\\[This.Math,Math.c\\] *:= *0 *;.*"
```

Listing 6.5.11: Regular Expression

```
const unique Math.c:Field int;
Heap[This.Math,Math.c] := 0 ;
```

Listing 6.5.12: Boogie Code Matched for Listing 6.5.10

6.5.5 Constant Declaration

```
(class Math()
  const c: Int32:=5;
class)
```

Listing 6.5.13: Constant Declaration

```
".*const *unique *Math.c *: *Field *int *;.*"
".*axiom *Math.c *== * *5 *;.*"
```

Listing 6.5.14: Regular Expression

```
const unique Math.c:Field int;
axiom Math.c == 5;
```

Listing 6.5.15: Boogie Code Matched for Listing 6.5.13

6.5.6 Ghost Object Initialization

```
(class Math()
  ghost obj c: Int32:=0;
class)
```

Listing 6.5.16: Ghost Object Declaration

```
".*const *unique *Math.c *: *Field *int *;.*"
".*Heap\\\[This.Math,Math.c\\] *:= *0 *;.*"
```

Listing 6.5.17: Regular Expression

```
const unique Math.c:Field int;
Heap[This.Math,Math.c] := 0 ;
```

Listing 6.5.18: Boogie Code Matched for Listing 6.5.16

Chapter 7

Conclusion and Suggested Future Work

7.1 Conclusion

We have successfully implemented, and tested the HARPO verifier. The HARPO verifier's implementation opens up a new way to verify the correctness of concurrent programs using the HARPO programming language. HARPO verifier is a significant contribution to the static verification research area. HARPO verifier is an addition to the list of verifiers such as *Chalice*, *Dafny*, *VCC*, and *HAVOC* all are designed based on Boogie verification methodology. The complete HARPO compiler ensures the correctness of generated C code and VHDL code if the HARPO verifier verifies it. Now HARPO programmers can write programs with no race conditions and index out of bound errors. We have addressed the frame problem and manually generated frame conditions for veri-

fication. The verification methodology and verifier design have setup new areas of research, such as dealing with verification of floating-point numbers, verification of user-defined functions, and *liveness* issues.

7.2 Future Work

Based on the research reported in this thesis, in the future, the HARPO project will be expanded to implement more features such as:

- Automatically inferring the loop invariants
- Addition of some language features, like functions and predicates into HARPO language
- Automatically inferring certain loop invariants to lessen the burden on the programmer in annotating their code
- Automatic generation of the frame conditions
- Implementation of dynamic objects creation in HARPO
- Implementation of VHDL and CUDA backends
- Implementation of the liveness property
- Integration of the HARPO verifier with the integrated development environment

Bibliography

- [1] Aircraft Accident Investigation Bureau, “Aircraft Accident Investigation Preliminary Report Ethiopian Airlines Group,” March 2019. [also available as: <https://leehamnews.com/wp-content/uploads/2019/04/Preliminary-Report-B737-800MAX-ET-AVJ.pdf> , accessed on 29-September-2019].
- [2] Wikipedia contributors, “Hitomi (satellite) — Wikipedia, the free encyclopedia,” 2019. [also available as: [https://en.wikipedia.org/wiki/Hitomi_\(satellite\)](https://en.wikipedia.org/wiki/Hitomi_(satellite)), accessed on 26 September – 2019].
- [3] “National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation, NASA Engineering and Safety Center,” January 2011.
- [4] “Failure of American Airlines Reservations System Grounds all Flights.” [also available as: <https://www.cnet.com/news/failure-of-american-airlines-reservations-system-grounds-all-flights/> , accessed on 25-September-2019].
- [5] N.G. Levenson and C.S. Turner, “An investigation of the therac-25 acci-

- dents,” *Computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [6] J.L. Lions, L. Beck, F. Jean-Luc, “Ariane 5 flight 501 failure report by the inquiry board, european space agency paris,” 1996.
- [7] K.M. Mead, Director Transportation Issues, “New Denver Airport, Impact of the Delayed Baggage System, United States, General Accounting Office, Washington, D.C.20548,” October 14 1994. [also available as: <https://www.gao.gov/assets/80/78935.pdf> , accessed on 19-September-2019].
- [8] C.A.R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [9] T.S. Norvell, “Language Design for CGRA Project. [Draft 10],” 2019.
- [10] T.S. Norvell, X. Li, D. Zhang, and M.A.T. Alam, “HARPO/L: A language for hardware/software codesign,” in *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, IEEE, 2008.
- [11] T.S. Norvell, “A Gainless Semantics for the HARPO,” in *Canadian Electrical and Computer Engineering Conference*, 2009.
- [12] F.Y. Ghalehjoogh, “Verification of HARPO Language, M.Eng. Thesis,” Memorial University of Newfoundland, 2013.
- [13] K.R.M. Leino, “This is Boogie 2,” *Manuscript KRML*, vol. 178, no. 131, 2008.
- [14] K.R.M. Leino, C. Flanagan, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, “Extended static checking for Java,” in *Proceedings of the ACM*

SIGPLAN 2002 Conference on Programming language design and implementation, pp. 234–245, 2002.

- [15] K. R. M. Leino and V. Wüstholtz, “The dafny integrated development environment,” *arXiv preprint arXiv:1404.6602*, 2014.
- [16] K. R. M. Leino, P. Muller, and J. Smans, “Verification of Concurrent Programs with Chalice,” 2009.
- [17] W. Schulte, “VCC: Contract-based Modular Verification of Concurrent C,” in *31st International Conference on Software Engineering, ICSE 2009*, IEEE Computer Society, January 2008.
- [18] O. Grumberg and M. Huth, eds., *A Reachability Predicate for Analyzing Low-Level Software*, (Berlin, Heidelberg), Springer Berlin Heidelberg, 2007.
- [19] C. Prause, R. Gerlich, and R. Gerlich, “Evaluating automated software verification tools,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 343–353, IEEE, 2018.
- [20] O. Hasan and S. Tahar, “Formal Verification Methods,” in *Encyclopedia of Information Science and Technology, Third Edition*, pp. 7162–7170, IGI Global, 2015.
- [21] J.M. Schumann, *Automated Theorem Proving in Software Engineering*. Springer Science & Business Media, 2001.
- [22] P. Ashar, A. Raghunathan, and S. Bhattacharya, “Verification of scheduling in the presence of loops using uninterpreted symbolic simulation,” June 3 2008. US Patent 7,383,166.

- [23] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.
- [24] L. Lamport, “The Hoare logic of concurrent programs,” *Acta Informatica*, vol. 14, no. 1, pp. 21–37, 1980.
- [25] J.C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, IEEE, 2002.
- [26] I.T. Kassios, “The Dynamic Frames Theory,” *Formal Aspects of Computing*, vol. 23, no. 3, pp. 267–288, 2011.
- [27] I.T. Kassios, “Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions,” in *FM 2006: Formal Methods*, (Berlin, Heidelberg), pp. 268–283, Springer Berlin Heidelberg, 2006.
- [28] J. Smans, B. Jacobs, and F. Piessens, “Implicit Dynamic Frames,” in *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*, pp. 1–12, 2008.
- [29] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*, vol. 11. Addison-Wesley Reading, 2000.
- [30] K.R.M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 348–370, Springer, 2010.
- [31] T.S. Norvell, “Annotations for Verification of HARPO/L [Draft version 0],” 2014.

- [32] T.S. Norvell, “Concurrent Software Verification with Explicit Transfer of Permission,” in *Newfoundland & Labrador Electrical and Computer Engineering Conference*, 2017.
- [33] I. Ahmed, T.S. Norvell and R. Venkatesan, “Verifying the Correctness of HARPO Programs,” in *Newfoundland & Labrador Electrical and Computer Engineering Conference*, 2018.

Appendices

Appendix A

Automated Translation Case Studies

A.1 Code Generation

A.1.1 Type Conversion

```
(class Math()
  obj a: Int8 := 0;
  obj b: Int16 := 0;
  obj c: Int32 := 0;
  obj d: Int64 := 0;
  obj e: Real16 := 0.0;
  obj f: Real32 := 0.0;
  obj g: Real64 := 0.0;
  obj h: Bool := true;
class)
```

Listing A.1.1: Variables Declaration of Various Types

```
".*const *unique *Math.[a-i] *: *Field *(int|real|bool
) *; .*"
```

```
".*Heap\\[This.Math,Math.[a-i]\\] *:= *(0|0.0|true|  
TODO) *;.*"
```

Listing A.1.2: Regular Expression

```
const unique Math.a:Field int;  
const unique Math.b:Field int;  
const unique Math.c:Field int;  
const unique Math.d:Field int;  
const unique Math.e:Field real;  
const unique Math.f:Field real;  
const unique Math.g:Field real;  
const unique Math.h:Field bool;  
Heap[This.Math,Math.a] := 0 ;  
Heap[This.Math,Math.b] := 0 ;  
Heap[This.Math,Math.c] := 0 ;  
Heap[This.Math,Math.d] := 0 ;  
Heap[This.Math,Math.e] := 0.0 ;  
Heap[This.Math,Math.f] := 0.0 ;  
Heap[This.Math,Math.g] := 0.0 ;  
Heap[This.Math,Math.h] := true ;
```

Listing A.1.3: Boogie Code Matched for Listing 6.5.19

A.1.2 Initialization with Boolean Expression

```
(class Math()  
  obj c: Bool:= true /\ false;  
class)
```

Listing A.1.4: Boolean Variable Declaration

```
".*const *unique *Math.c *: *Field *bool *;.*"  
.Heap\\[This.Math,Math.c\\] *:= *true *\&\&\& *false  
*;.*"
```

Listing A.1.5: Regular Expression

```
const unique Math.c:Field bool;  
Heap[This.Math,Math.c] := true && false ;
```

Listing A.1.6: Boogie Code Matched for Listing 6.5.22

A.1.3 Chained Initialization Expression

```
(class Math()
  obj c: Bool := false /\ true \/ true;
class)
```

Listing A.1.7: Chained Expression

```
".*const *unique *Math.c *: *Field *bool *;.*"
".*Heap\\\[This.Math,Math.c\\] *:= *false *\&\&\& *\&\& *
true *\&\&\& *true *;.*"
```

Listing A.1.8: Regular Expression

```
const unique Math.c:Field bool;
Heap [This.Math,Math.c] := false && true || true ;
```

Listing A.1.9: Boogie Code Matched for Listing 6.5.25

A.1.4 Initialization Expressions

```
(class Math()
  obj a: Int8 := 5+9;
  obj b: Int16 := 8*6+9;
  obj c: Int32 := 2000 div 10;
  obj d: Int64 := 4000000*8;
  obj e: Real16 := 2000.035/89.08;
  obj f: Real32 := 96.0*85.0-42.3+3000.2*23.220014;
  obj g: Real64 := 0.00000000005658/96.22;
  obj h: Bool := true /\ false \/ true;
  obj i: Bool := (5 _< 89) => (4 _< 89)
  obj j: Bool := (4 _< 89) <= (5 _< 89)
  obj k: Bool := (5 _< 89) <=> (4 _< 89)
  obj l: Bool := ~ (true)
  obj m: Bool := (5 = 89)
  obj n: Bool := (5 _< 89)
  obj o: Bool := (5 >_ 89)
  obj p: Bool := (5 ~= 89)
  obj q: Bool := (5 < 89)
  obj r: Bool := (5 > 89)
  obj s: Int32 := 1000 mod 20
  obj t: Bool := (5 _< 89)
  obj u : Int32 := a - 20
```

```
class)
```

Listing A.1.10: Initialization Expressions of Various Types

```
".*const *unique *Math.[a-u] *: *Field *(int|real|bool  
) *;.*"  
".*Heap\\[This.Math,Math.[a-u]\\] *:= *.*;.*
```

Listing A.1.11: Regular Expression

```

const unique Math.a:Field int;
const unique Math.b:Field int;
const unique Math.c:Field int;
const unique Math.d:Field int;
const unique Math.e:Field real;
const unique Math.f:Field real;
const unique Math.g:Field real;
const unique Math.h:Field bool;
const unique Math.i:Field bool;
const unique Math.j:Field bool;
const unique Math.k:Field bool;
const unique Math.l:Field bool;
const unique Math.m:Field bool;
const unique Math.n:Field bool;
const unique Math.o:Field bool;
const unique Math.p:Field bool;
const unique Math.q:Field bool;
const unique Math.r:Field bool;
const unique Math.s:Field int;
const unique Math.t:Field bool;
const unique Math.u:Field int;
Heap[This.Math,Math.a] := 5 + 9 ;
Heap[This.Math,Math.b] := 8 * 6 + 9 ;
Heap[This.Math,Math.c] := 2000 / 10 ;
Heap[This.Math,Math.d] := 4000000 * 8 ;
Heap[This.Math,Math.e] := 2000.035 / 89.08 ;
Heap[This.Math,Math.f] := 96.0 * 85.0 - 42.3 + 3000.2 *
    23.220014 ;
Heap[This.Math,Math.g] := 0.00000000005658 / 96.22 ;
Heap[This.Math,Math.h] := true && false || true ;
Heap[This.Math,Math.i] := 5<=89 ==> 4<=89 ;
Heap[This.Math,Math.j] := 4<=89 ==> 5<=89 ;
Heap[This.Math,Math.k] := 5<=89 == 4<=89 ;
Heap[This.Math,Math.l] := !true ;
Heap[This.Math,Math.m] := 5==89 ;
Heap[This.Math,Math.n] := 5<=89 ;
Heap[This.Math,Math.o] := 5>=89 ;
Heap[This.Math,Math.p] := 5!=89 ;
Heap[This.Math,Math.q] := 5<89 ;
Heap[This.Math,Math.r] := 5>89 ;
Heap[This.Math,Math.s] := 1000 % 20 ;
Heap[This.Math,Math.t] := 5<=89 ;
Heap[This.Math,Math.u] := Heap[This.Math,Math.a] - 20 ;

```

Listing A.1.12: Boogie Code Matched for Listing 6.5.28

A.1.5 Class Claim Specification

```
class Math()
claim c@1.0, d@0.000003, e@0.1000003000000007
obj c: Int32:=0;
obj d: Int32:=0;
obj e: Int32:= 1;
(thread(*t0*)
  assert c>9
thread)
class)
```

Listing A.1.13: Class Claim Specification

```
".*assert IsValidPermission\\(([0-9]+.[0-9]+)\\)\"",
".*Permission\\[This.Math,Math.(c|d|e) *\\] *= *
  Permission\\[This.Math,Math.(c|d|e) *\\] *\\+
  *[0-9]+.[0-9]+) *;.*",
"assert *0.0 *<= *Permission\\[This.Math,Math.(c|d|e)
\\] *\\&\\& *Permission\\[This.Math,Math.(c|d|e)
\\] *<= *1.0 *;.*"
```

Listing A.1.14: Regular Expressions

```
Permission[This.Math,Math.c] := Permission[This.Math,Math.
  c] + 1.0;
Permission[This.Math,Math.d] := Permission[This.Math,Math.
  d] + 0.000003;
Permission[This.Math,Math.e] := Permission[This.Math,Math.
  e] + 0.1000003000000007;
assert 0.0 <= Permission[This.Math,Math.c] && Permission[
  This.Math,Math.c]<=1.0;
assert 0.0 <= Permission[This.Math,Math.d] && Permission[
  This.Math,Math.d]<=1.0;
assert 0.0 <= Permission[This.Math,Math.e] && Permission[
  This.Math,Math.e]<=1.0;
```

Listing A.1.15: Boogie Code Matched for Listing 6.5.31

A.1.6 Assert Command

```
(class Math()
  obj c: Int32:=0;
  (thread(*t0*)
```

```

    assert (c>9 /\ c<100) /\ (c ~=~ 20 /\ c ~=~ 60)
  thread)
class)

```

Listing A.1.16: Assert Command

```

".*assert +isInt32\\((9|100|20|60|Heap\\[This.Math,
  Math.c\\])\\).*",
".*assert +Permission\\[This.Math,Math.c\\] *>
  *0.0.*",
".*assert *Heap\\[This.Math,Math.c\\] *> *9 *\\&\\& *
  Heap\\[This.Math,Math.c\\] *< *100 *\\&\\& *Heap
  \\[This.Math,Math.c\\] *!= *20 *\\&\\& *Heap\\[
  This.Math,Math.c\\] *!= *60 *;.*"

```

Listing A.1.17: Regular Expressions

```

assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(9);
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(100);
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(20);
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(60);
assert Permission[This.Math,Math.c] > 0.0 ;
assert Heap[This.Math,Math.c] > 9 && Heap[This.Math,Math.c]
  ] < 100 && Heap[This.Math,Math.c] != 20 && Heap[This.
  Math,Math.c] != 60;

```

Listing A.1.18: Boogie Code Matched for Listing 6.5.34

A.1.7 Local Variable Initialization

```

(class Math()
  obj c: Int32:=0;
(thread(*t0*)
  obj b: Int32 := c+4;
thread)
class)

```

Listing A.1.19: Local Variable Declaration

```

".*assert +isInt32\\((Heap\\[This.Math,Math.c\\]|4)\\)
.*",
".*Heap\\[This.Math,Math.t#0.b\\] *:= *Heap\\[This.
Math,Math.c\\] *\\+ *4 *;.*"

```

Listing A.1.20: Regular Expressions

```

assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(4);
Heap[This.Math,Math.t#0.b] := Heap[This.Math,Math.c] + 4;

```

Listing A.1.21: Boogie Code Matched for Listing 6.5.37

A.1.8 Bool Type Local Variable

```

(class Math()
  obj c: Bool := false;
(thread(*t0*)
  obj b: Bool := c /\ false;
thread)
class)

```

Listing A.1.22: Bool Type Local Variable

```

".*assert +isBool\\((Heap\\[This.Math,Math.c\\]|false)
\\).*",
".*Heap\\[This.Math,Math.t#0.b\\] *:= *Heap\\[This.
Math,Math.c\\] *\\&\\& *false *;.*"

```

Listing A.1.23: Regular Expressions

```

assert isBool(Heap[This.Math,Math.c]);
assert isBool(false);
Heap[This.Math,Math.t#0.b] := Heap[This.Math,Math.c] &&
false;

```

Listing A.1.24: Boogie Code Matched for Listing 6.5.40

A.1.9 Assume Command

```

(class Math()
  obj c: Int32:=0;

```

```

(thread(*t0*)
  obj d:Real32 := 34.98765;
  assume c>9 /\ d < 35
thread)
class)

```

Listing A.1.25: Assume Command

```

".*assert *isReal64\\(34.98765\\) *;.*",
".*Heap\\[This.Math,Math.c\\] *:= *34.98765 *;.*",
".*assert *isInt32\\((9|35|Heap\\[This.Math,Math.c\\])
  \\) *;.*",
".*assert *isReal32\\(Heap\\[This.Math,Math.t#0.d
  \\]\\) *;.*",
".*assert *Permission\\[This.Math,Math.c\\] *> 0.0
  *\\&\\& *Permission\\[This.Math,Math.t#0.d\\] *>
  *0.0 *;.*",
".*assume *Heap\\[This.Math,Math.c\\] *> 9 *\\&\\& *
  Heap\\[This.Math,Math.t#0.d\\] *< *35 *;.*"

```

Listing A.1.26: Regular Expressions

```

assert isReal64(34.98765);
//Expression Definedness Start
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(9);
assert isInt32(35);
assert isReal32(Heap[This.Math,Math.t#0.d]);
assert Permission[This.Math,Math.c] > 0.0 && Permission[
  This.Math,Math.t#0.d] > 0.0 ;
assume Heap[This.Math,Math.c] > 9 && Heap[This.Math,Math.t
#0.d] < 35;

```

Listing A.1.27: Boogie Code Matched for Listing 6.5.43

A.1.10 If Command

```

(class Math()
  obj c: Int8:= 20;
  (thread(*t0*)
    (if(c _< 21) then obj b: Bool := false;)
  thread)
class)

```

Listing A.1.28: If Command

```

".*assert *isInt8\\(Heap\\[This.Math,Math.c\\]\\)
*;.*",
".*assert *isInt32\\(21\\) *;.*",
".*assert *Heap\\[This.Math,Math.c\\] *> *0.0 *;.*",
".*if *\\\(.*\\) *\n* *\{\ *\n*([^\n])*\}.*"

```

Listing A.1.29: Regular Expressions

```

assert isInt8(Heap[This.Math,Math.c]);
assert isInt32(21);
assert Heap[This.Math,Math.c] > 0.0 ;if(Heap[This.Math,
Math.c] <= 21)
assert Heap[This.Math,Math.c] > 0.0 ;if(Heap[This.Math,
Math.c] <= 21)
{
    assert isBool(false);
    Heap[This.Math,Math.t#0.b] := false;
}

```

Listing A.1.30: Boogie Code Matched for Listing Listing 6.5.46

A.1.11 While Command

```

(class Math()
  obj c: Int32:=0;
  (thread(*t0*)
    (while(true)
      do
        c := c+1;
      while)
    thread)
class)

```

Listing A.1.31: While Command

```

".*while *\\\( *true *\\) *\n* *invariant *\\\(.*?\\) *;
*\n* *\{\ *\n*([^\n])*\}.*"

```

Listing A.1.32: Regular Expression

```

while( true)
  invariant (forall<x> r: Ref, f: Field x :: 0.0 <=
    Permission[r,f] && Permission[r,f] <= 1.0);
{

```

```

//Check Assignment Definedness
//Expression Definedness Start
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(Heap[This.Math,Math.c]);
assert isInt32(1);
//Expression Definedness Ends
assert Permission[This.Math,Math.c] == 1.0 ;
assert Permission[This.Math,Math.c] > 0.0;
//Check Assignment Definedness Ends
//Assignment Command
Heap[This.Math,Math.c] := Heap[This.Math,Math.c] + 1;
//Assignment Command Ends
}
}
//end of Thread Procedure

```

Listing A.1.33: Boogie Code Matched for 6.5.49

A.1.12 For Command

```

(class Math()
  obj c: Int32:=20;
  obj f: Int64 := 0;
  (thread(*t0*)
    (for x : c do
      f := x+1
      for)
    thread)
  class)

```

Listing A.1.34: For Command

```
".*while *\\(.*)\\) *\\n* *\\{ *\\n*([^\n}]*) .*
```

Listing A.1.35: Regular Expression

```

while ( Heap[This.Math,Math.t#0.#f1] < Heap[This.Math,Math
.c] )
{
//Check Assignment Definedness
//Expression Definedness Start
assert isInt64(Heap[This.Math,Math.f]);
assert isInt32(1);
//Expression Definedness Ends

```

```

assert Permission[This.Math.Math.f] == 1.0 ;
assert Permission[This.Math.Math.t#0.#f1.x] > 0.0;
//Check Assignment Definedness Ends
//Assignment Command
Heap[This.Math.Math.f] := Heap[This.Math.Math.t#0.#f1.x]
+ 1;
//Assignment Command Ends
Heap[This.Math.Math.t#0.#f1] := Heap[This.Math.Math.t#0.#f1] + 1;
}
}
//end of Thread Procedure

```

Listing A.1.36: Boogie Code Matched for Listing 6.5.52

A.1.13 Call Command

```

(class Math()
  obj a : Int32 := 21;
  obj b : Int32 := 31;
  obj sum : Int32 := 0;
  proc add()
    pre a > 20 /\ b > 10
    post sum > 30
    (thread (*t0*)
      (accept add()
        sum := a+b;
        accept)
      thread)
    (thread (*t1*)
      add()
      thread)
  class)

```

Listing A.1.37: Call Command

```

".*goto add; *\n* *[^\n]* *\n* *\n*end of Thread
Procedure.*"

```

Listing A.1.38: Regular Expression

```

procedure Math.t#0 (This.Math : Ref)
requires dtype(This.Math) <: Math;
modifies Heap;

```

```

modifies LockPermission;
{
    var local_Permission : PermissionType where (forall <x> r
        : Ref, f: Field x :: Permission[r,f] == 0.0);
    var local_oldPermission: PermissionType;
    var LockPermission : PermissionType where (forall <x> r:
        Ref, f: Field x :: LockPermission[r,f] == 0.0);
    var local_ArrayPermission : ArrayPermissionType where (
        forall <x> r: ArrayRef x, f: int :: ArrayPermission[r,
        f] == 0.0);
    var local_OldArrayPermission: PermissionType;
    var ArrayLockPermission : ArrayPermissionType where (
        forall <x> r: ArrayRef x, f: int :: ArrayLockPermission[r,f] == 0.0);
    var oldHeap, preHeap, tempHeap: HeapType;
    var oldArrayHeap, preArrayHeap, tempArrayHeap:
        ArrayHeapType;
    goto add;
    add:
    oldHeap := Heap;
    havoc Heap;
    assume Heap[This.Math.Math.a] > 20 && Heap[This.
    Math, Math.b] > 10;
    assert isInt32(Heap[This.Math.Math.sum]);
    assert isInt32(Heap[This.Math.Math.a]);
    assert isInt32(Heap[This.Math.Math.b]);
    assert Permission[This.Math.Math.sum] == 1.0 ;
    assert Permission[This.Math.Math.a] > 0.0 &&
        Permission[This.Math.Math.b] > 0.0;
    Heap[This.Math.Math.sum] := Heap[This.Math.Math.a]
    +
        Heap[This.Math.Math.b];
    assert Permission[This.Math.Math.sum] > 0.0;
    assert Heap[This.Math.Math.sum] > 30;
    goto end;
end:
}

procedure Math.t#1 (This.Math : Ref)
requires dtype(This.Math) <: Math;
modifies Permission;
modifies LockPermission;
{
    var local_Permission : PermissionType where (forall <x> r
        : Ref, f: Field x :: Permission[r,f] == 0.0);
    var local_oldPermission: PermissionType;
}

```

```

var LockPermission : PermissionType where (forall <x> r:
    Ref, f: Field x :: LockPermission[r,f] == 0.0);
var local_ArrayPermission : ArrayPermissionType where (
    forall <x> r: ArrayRef x, f: int :: ArrayPermission[r,
    f] == 0.0);
var local_OldArrayPermission: PermissionType;
var ArrayLockPermission : ArrayPermissionType where (
    forall <x> r: ArrayRef x, f: int :::
    ArrayLockPermission[r,f] == 0.0);
var oldHeap, preHeap, tempHeap: HeapType;
var oldArrayHeap, preArrayHeap, tempArrayHeap:
    ArrayHeapType;
preHeap := Heap;
preArrayHeap := ArrayHeap;
}//end of Thread Procedure

```

Listing A.1.39: Boogie Code Matched for Listing 6.5.55

A.2 Counter

A concurrent HARPO example where thread $*t0*$ tries to make an increment into the counter variable. We have tested different variants of *counter* class and checked all the from the HARPO verifier. Class diagram in Figure A.1 and message sequence in Figure A.1 are visual representations of *Counter* class and $*t0*$'s message sequence.

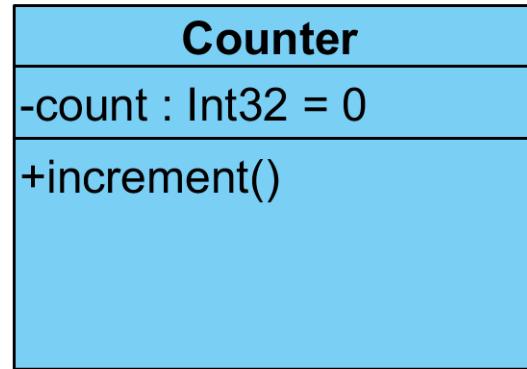


Figure A.1: Counter Class Diagram

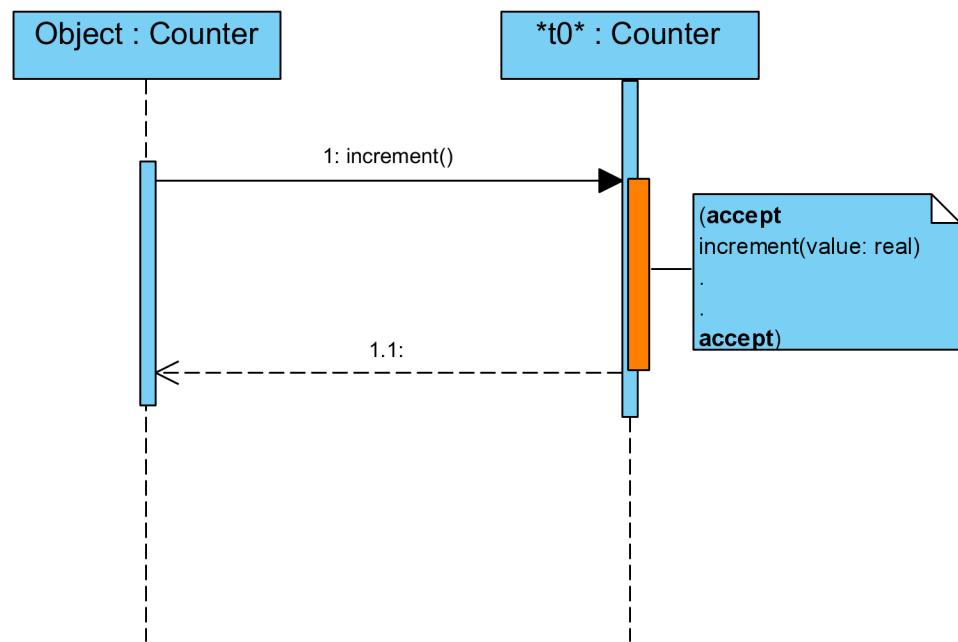


Figure A.2: Counter Class Message Sequence Diagram

A.2.1 Counter Version 0

```
(class Counter()
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
obj count: Int32 := 0
(thread (*t0*)
(while true
  do
    (accept increment()
      (with this
        do
          count := count+1
        with)
      accept)
    while)
  thread)
class)
```

Listing A.2.1: Counter Version 0

```
// Prelude begin
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;

var Heap:HeapType;

type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;
var ArrayHeap:ArrayHeapType;

type Perm = real;
type PermissionType = <x> [Ref,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;

function Length<x>(Field (ArrayRef x)) returns (int);
```

```

const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;

const unique min16:int;
axiom min16 == -32768;
const unique max16:int;
axiom max16 == 32767;

const unique min32:int;
axiom min32 == -2147483648;
const unique max32:int;
axiom max32 == 2147483647;

const unique min64:int;
axiom min64 == -9223372036854775808;
const unique max64:int;
axiom max64== 9223372036854775807;

function Isint8(int) returns (bool);
axiom (forall x:int :: Isint8(x) <==> min8 <=x&&x<=max8);

function Isint16(int) returns (bool);
axiom (forall x:int :: Isint16(x) <==> min16 <=x&&x<=max16
);

function Isint32(int) returns (bool);
axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
);

function Isint64(int) returns (bool);
axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
max64);

const unique minPer : Perm;
axiom minPer == 0.0;
const unique maxPer : Perm;
axiom maxPer == 1.0;

function IsValidPermission(Perm) returns (bool);
axiom (forall x: Perm:: IsValidPermission(x) <==> minPer
<= x && x <= maxPer);

const unique TenPow16 : real;

```



```

type ClassName;
function dtype(Ref) returns (ClassName);

// Prelude ends

const unique Counter:ClassName;

const unique Counter.count:Field int;
procedure Counter.constructor(This_Counter:Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
{
    var Permission : PermissionType where (forall <x> r:
    Ref, f: Field x :: Permission[r,f] == 0.0);
    var oldHeap:HeapType;
    havoc Heap;
    //Claim
    Permission[This_Counter,Counter.count] := Permission[
    This_Counter,Counter.count] + 0.5;
    assert (forall<x> r: Ref, f: Field x :: Permission[r,f]
    ] <= 1.0);

    //Initialize Heap
    Heap[This_Counter,Counter.count] := 0;

    //Class Invariant
    assert Permission[This_Counter,Counter.count] > 0.0
    && Heap[This_Counter,Counter.count] >= 0;

}
procedure Counter.t#0 (This_Counter : Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
modifies LockPermission;
{
    var Permission: PermissionType where (forall<x> r: Ref
    , f: Field x :: Permission[r,f] == 0.0);
    var oldPermission: PermissionType where (forall<x> r:
    Ref, f: Field x :: oldPermission[r,f] == 0.0);
    var oldHeap, preHeap, tempHeap: HeapType;

```

```

var this_lock: Ref;
while( true)
{
    goto increment;
increment:

//Taking Permission(s)
oldPermission := Permission;
if(Permission[This_Counter,Counter.count] == 0.0)
{
    havoc tempHeap;
    Heap[This_Counter,Counter.count] := tempHeap[
        This_Counter,Counter.count];
    Permission[This_Counter,Counter.count] :=
        Permission[This_Counter,Counter.count]+0.5;
}
//Pre Condition(s)
oldHeap := Heap;
havoc Heap;
assume Heap[This_Counter,Counter.count] >= 0;

//Method Implementation
this_lock:=This_Counter;
preHeap := Heap;
havoc tempHeap;
havoc LockPermission;

//Assert definedness of object invariant and assume
//object invariance
assert LockPermission[this_lock,Counter.count] >
0.0;
assume LockPermission[this_lock,Counter.count] > 0.0
&& Heap[This_Counter,Counter.count] >= 0;

//Check Assignment Definedness
assert LockPermission[This_Counter,Counter.count] +
Permission[This_Counter,Counter.count] == 1.0 ;
assert LockPermission[This_Counter,Counter.count] >
0.0;
assert Permission[This_Counter,Counter.count] > 0.0;
//Check Assignment Definedness Ends

//Assignment Command
Heap[This_Counter,Counter.count] := Heap[
This_Counter,Counter.count] + 1;

```

```

//Assignment Command Ends

    //Assert definedness of object invariant and assume
    object invariance

        assert LockPermission[this_lock,Counter.count] >
    0.0;
        assert LockPermission[this_lock,Counter.count] > 0.0
        && Heap[This_Counter,Counter.count]           >= 0;
        //Post Condition(s)
        assert Permission[This_Counter,Counter.count] > 0.0;

        assert Heap[This_Counter,Counter.count] > 0;

        //Giving Permissions(s)
        assert Permission[This_Counter,Counter.count] >=
    0.5;
        Permission[This_Counter,Counter.count] := Permission
    [This_Counter,Counter.count]           - 0.5;
        goto end;
    end:
}
}//end of Thread Procedure

```

Listing A.2.2: Counter v0 - Boogie Output of Code Generator

A.2.2 Counter Version 1: without class invariant

```

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

```

```
    accept)
  while)
thread)
class)
```

Listing A.2.3: Counter Version 1

```
// Prelude begin
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;

var Heap:HeapType;

type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;
var ArrayHeap:ArrayHeapType;

type Perm = real;
type PermissionType = <x> [Ref,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;

function Length<x>(Field (ArrayRef x)) returns(int);

const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;

const unique min16:int;
axiom min16 == -32768;
const unique max16:int;
axiom max16 == 32767;

const unique min32:int;
axiom min32 == -2147483648;
const unique max32:int;
axiom max32 == 2147483647;

const unique min64:int;
axiom min64 == -9223372036854775808;
const unique max64:int;
axiom max64== 9223372036854775807;
```

```

function Isint8(int) returns (bool);
axiom (forall x:int :: Isint8(x) <==> min8 <=x&&x<=max8);

function Isint16(int) returns (bool);
axiom (forall x:int :: Isint16(x) <==> min16 <=x&&x<=max16
);

function Isint32(int) returns (bool);
axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
);

function Isint64(int) returns (bool);
axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
max64);

const unique minPer : Perm;
axiom minPer == 0.0;
const unique maxPer : Perm;
axiom maxPer == 1.0;

function IsValidPermission(Perm) returns (bool);
axiom (forall x: Perm:: IsValidPermission(x) <==> minPer
<= x && x <= maxPer);
const unique TenPow16 : real;
axiom TenPow16 == 1000000000000000.0; // 10 ^ 16
const unique TenPow32 : real;
axiom TenPow32 == 10000000000000000000000000000000.0; //
10 ^ 32
const unique TenPow64 : real;
axiom TenPow64 == 10000000000000000000000000000000000000000000.0; // 10 ^ 64

const unique minReal16:real;
axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
const unique maxReal16:real;
axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

const unique minReal32:real;
axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
3.402823466 E - 38
const unique maxReal32:real;
axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
//3.402823465 E + 38

const unique minReal64:real;

```

```

axiom minReal64 == -1.7976931348623157 / (TenPow64*
    TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0);
// -1.7976931348623157 E - 308
const unique maxReal64:real;
axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64
    *TenPow64*TenPow64*TenPow32*TenPow16*1000.0); //
1.7976931348623156 E + 308

function isReal16(real) returns (bool);
axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
    x <= maxReal16);

function isReal32(real) returns (bool);
axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
    x <= maxReal32);

function isReal64(real) returns (bool);
axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
    x <= maxReal64);

function isBool(bool) returns (bool);
axiom (forall x: bool :: isBool(x) <==> x == true || x ==
    false);

type ClassName;
function dtype(Ref) returns (ClassName);

// Prelude ends

const unique Counter:ClassName;
const unique Counter.count:Field int;
procedure Counter.constructor(This_Counter:Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
{
    var Permission : PermissionType where (forall <x> r:
        Ref, f: Field x :: Permission[r,f] == 0.0);
    var oldHeap:HeapType;
    havoc Heap;
    //Claim
    Permission[This_Counter,Counter.count] := Permission[
        This_Counter,Counter.count] + 0.5;
    assert (forall<x> r: Ref, f: Field x :: Permission[r,f]

```

```

] <= 1.0);

//Initialize Heap
Heap[This_Counter,Counter.count] := 0;
}

procedure Counter.t#0 (This_Counter : Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
modifies LockPermission;
{
    var Permission: PermissionType where (forall<x> r: Ref
, f: Field x :: Permission[r,f] == 0.0);
    var oldPermission: PermissionType where (forall<x> r:
Ref, f: Field x :: oldPermission[r,f] == 0.0);
    var oldHeap, preHeap, tempHeap: HeapType;

    while( true)
        //invariant forall<x> r: Ref, f: Field x :: 0.0 <=
Permission[r,f] <= 1.0;
    {
        goto increment;
        increment:

        //Taking Permission(s)
        oldPermission := Permission;
        if(Permission[This_Counter,Counter.count] == 0.0)
        {
            havoc tempHeap;
            Heap[This_Counter,Counter.count] := tempHeap [
                This_Counter,Counter.count];
            Permission[This_Counter,Counter.count] :=
Permission [This_Counter,Counter.count]+0.5;
        }
        //Pre Condition(s)
        oldHeap := Heap;
        havoc Heap;
        assume Heap[This_Counter,Counter.count] >= 0;

        //Method Implementation
        //Check Assignment Definedness
        assert Permission[This_Counter,Counter.count
]==1.0;
        assert Permission[This_Counter,Counter.count] >
0.0;
    }
}

```

```

//Check Assignment Definedness Ends
//Assignment Command
Heap[This_Counter,Counter.count] := Heap[
    This_Counter,Counter.count] + 1;
//Assignment Command Ends
//Post Condition(s)
assert Permission[This_Counter,Counter.count] >
0.0;
assert Heap[This_Counter,Counter.count] > 0;
//Giving Permissions(s)
assert Permission[This_Counter,Counter.count] >=
0.5;
Permission[This_Counter,Counter.count] :=
Permission[This_Counter,Counter.count] -
0.5;
goto end;
end:
}
}//end of Thread Procedure

```

Listing A.2.4: Counter v1-Boogie Output of Code Generator

A.2.3 Counter Version 2: without locking mechanism

```

(class Counter()
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
  obj count: Int32 := 0
(thread (*t0*)
  (while true
    do
      (accept increment()
        count := count+1
      accept)
    while)
  thread)
class)

```

Listing A.2.5: Counter Example

```

// Prelude begin
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;

var Heap:HeapType;

type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;
var ArrayHeap:ArrayHeapType;

type Perm = real;
type PermissionType = <x> [Ref,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;

function Length<x>(Field (ArrayRef x)) returns(int);

const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;

const unique min16:int;
axiom min16 == -32768;
const unique max16:int;
axiom max16 == 32767;

const unique min32:int;
axiom min32 == -2147483648;
const unique max32:int;
axiom max32 == 2147483647;

const unique min64:int;
axiom min64 == -9223372036854775808;
const unique max64:int;
axiom max64== 9223372036854775807;

function Isint8(int) returns (bool);
axiom (forall x:int :: Isint8(x) <=> min8 <=x&&x<=max8);

function Isint16(int) returns (bool);
axiom (forall x:int :: Isint16(x) <=> min16 <=x&&x<=max16)

```

```

);

function Isint32(int) returns (bool);
axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
);

function Isint64(int) returns (bool);
axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
max64);

const unique minPer : Perm;
axiom minPer == 0.0;
const unique maxPer : Perm;
axiom maxPer == 1.0;

function IsValidPermission(Perm) returns (bool);
axiom (forall x: Perm:: IsValidPermission(x) <==> minPer
<= x && x <= maxPer);

type ClassName;
function dtype(Ref) returns (ClassName);

const unique TenPow16 : real;
axiom TenPow16 == 1000000000000000.0; // 10 ^ 16
const unique TenPow32 : real;
axiom TenPow32 == 10000000000000000000000000000000.0; //
10 ^ 32
const unique TenPow64 : real;
axiom TenPow64 == 100000000000000000000000000000000000000000000.0; //
00000000000000000000000000000000.0; // 10 ^ 64

const unique minReal16:real;
axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
const unique maxReal16:real;
axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

const unique minReal32:real;
axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
3.402823466 E - 38
const unique maxReal32:real;
axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
//3.402823465 E + 38

const unique minReal64:real;
axiom minReal64 == -1.7976931348623157 / (TenPow64*

```

```

TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0) ;
// -1.7976931348623157 E - 308
const unique maxReal64:real;
axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64
*TenPow64*TenPow64*TenPow32*TenPow16*1000.0); //
1.7976931348623156 E + 308

function isReal16(real) returns (bool);
axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
x <= maxReal16);

function isReal32(real) returns (bool);
axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
x <= maxReal32);

function isReal64(real) returns (bool);
axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
x <= maxReal64);

function isBool(bool) returns (bool);
axiom (forall x: bool :: isBool(x) <==> x == true || x ==
false);

// Prelude ends

const unique Counter:ClassName;

const unique Counter.count:Field int;
procedure Counter.constructor(This_Counter:Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
{
    var Permission : PermissionType where (forall <x> r:
Ref, f: Field x :: Permission[r,f] == 0.0);
    var oldHeap:HeapType;
    havoc Heap;
    //Claim
    Permission[This_Counter,Counter.count] := Permission[
This_Counter,Counter.count] + 0.5;
    assert (forall<x> r: Ref, f: Field x :: Permission[r,f]
] <= 1.0);

    //Initialize Heap
    Heap[This_Counter,Counter.count] := 0;
}

```

```

//Class Invariant
assert Permission[This_Counter,Counter.count] > 0.0
&& Heap[This_Counter,Counter.count] >= 0;

}

procedure Counter.t#0 (This_Counter : Ref)
requires dtype(This_Counter) <: Counter;
modifies Heap;
modifies LockPermission;
{
    var Permission: PermissionType where (forall<x> r: Ref
, f: Field x :: Permission[r,f] == 0.0);
    var oldPermission: PermissionType where (forall<x> r:
Ref, f: Field x :: oldPermission[r,f] == 0.0);
    var oldHeap, preHeap, tempHeap: HeapType;

    while( true)
    {
        goto increment;
        increment:

        //Taking Permission(s)
        oldPermission := Permission;
        if(Permission[This_Counter,Counter.count] == 0.0)
        {
            havoc tempHeap;
            Heap[This_Counter,Counter.count] := tempHeap[
This_Counter,Counter.count];
            Permission[This_Counter,Counter.count] :=
Permission[This_Counter,Counter.count]+0.5;
        }

        //Pre Condition(s)
        oldHeap := Heap;
        havoc Heap;
        assume Heap[This_Counter,Counter.count] >= 0;

        //Method Implementation
        //Check Assignment Definedness
        assert Permission[This_Counter,Counter.count] ==
1.0;
        assert Permission[This_Counter,Counter.count] >
0.0;
    }
}

```

```

//Check Assignment Definedness Ends
//Assignment Command
Heap[This_Counter,Counter.count] := Heap[
This_Counter,Counter.count] + 1;
//Assignment Command Ends
//Post Condition(s)
assert Permission[This_Counter,Counter.count] > 0.0;
assert Heap[This_Counter,Counter.count] > 0;

//Giving Permissions(s)
assert Permission[This_Counter,Counter.count] >= 0.5;
Permission[This_Counter,Counter.count] := Permission[
This_Counter,Counter.count] - 0.5;
goto end;
end:
}
}//end of Thread Procedure

```

Listing A.2.6: Counter v2-Boogie Output of Code Generator

A.3 Buffer: reading and writing

The class diagram in Figure A.3 and message sequence in Figure A.4 are visual representations of properties and internal operations of *Buffer* class.

```

(class Buffer()
  proc deposit(in value : Real64)
  proc fetch(out value : Real64)
  const size : Int32 := 10
  obj buf : Real64[size] := (for i:size do 0 for)
  obj front : Int32 :=0
  obj rear : Int32 :=0
  obj full : Int32 :=0
(thread (*t0*) claim front, rear, full, {i:{0,..size}.buf
  [i]})
  (while true
    invariant canWrite(front) /\ \
      canWrite(rear) /\ \
      canWrite(full) /\ \
      canWrite({i:{0,..size}.buf[i]}) )
  invariant (0 _< front /\ front < size) /\ \

```

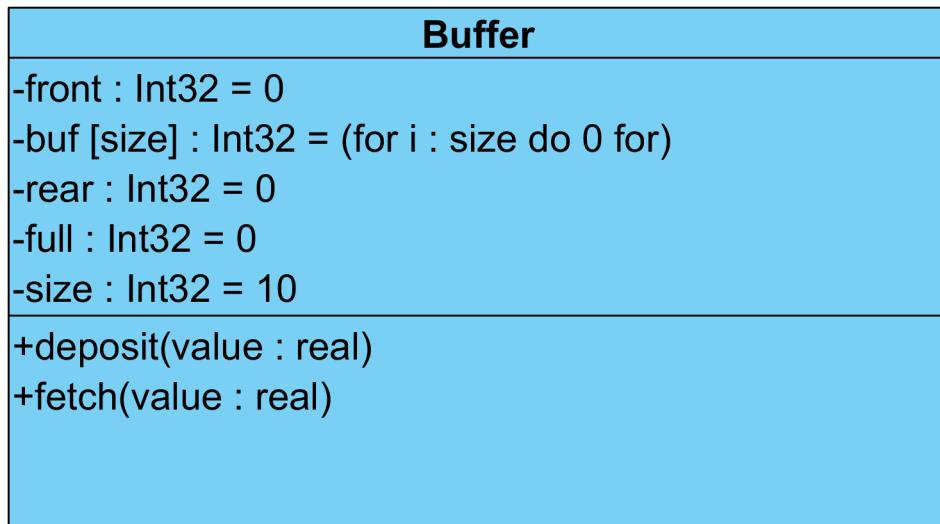


Figure A.3: Buffer Class Diagram

```

        (0 _< rear /\ rear < size) /\ 
        (_< full /\ full < size)
invariant ((front + full) mod size = rear)
do
  accept deposit(in value : Real64) when (full < size)
    buf[rear] := value
    rear := (rear+1) mod size
    full := full+1
  |
    fetch(out ovalue: Real64) when (0 < full)
    ovalue := buf[front]
    front := (front+1) mod size
    full := full-1
  accept)
  while)
  thread)
class)

```

Listing A.3.1: Buffer Class

```

// Prelude
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;
var Heap:HeapType;
type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;

```

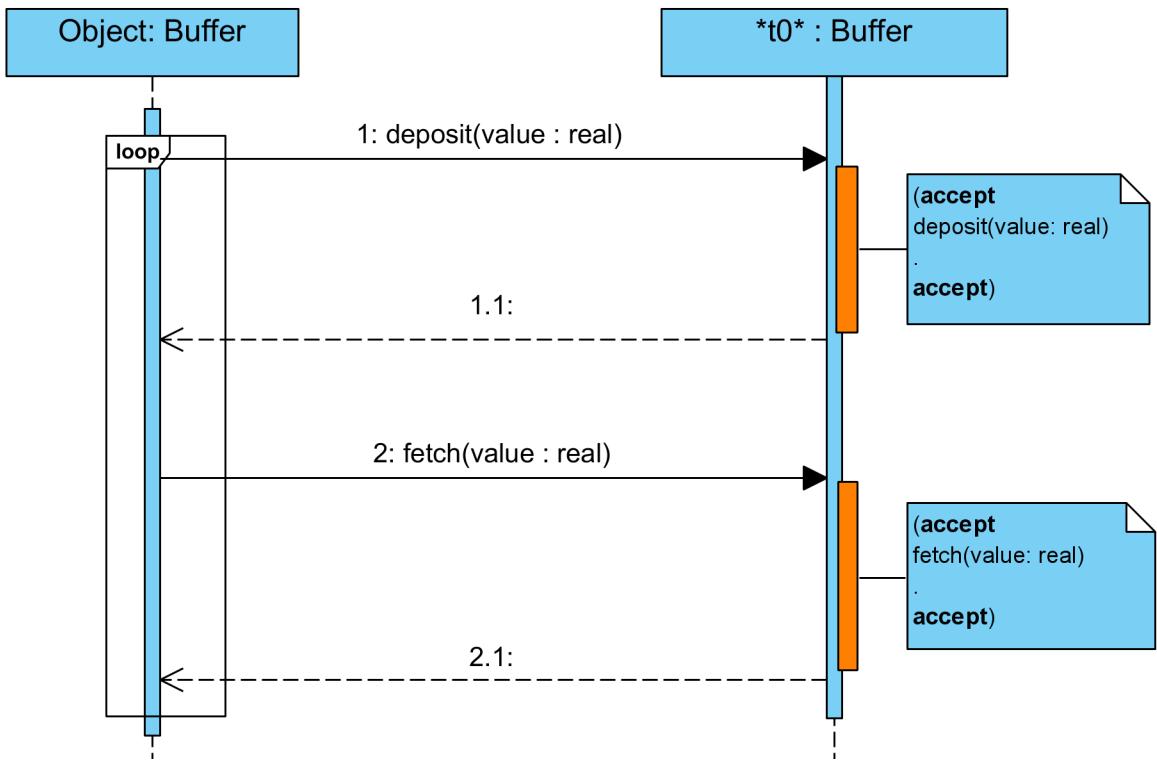


Figure A.4: Buffer Class Message Sequence Diagram

```

var ArrayHeap:ArrayHeapType;
type Perm = real;
type PermissionType = <x> [Ref ,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;
function Length<x>(Field (ArrayRef x))returns(int);
const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;
const unique min16:int;
axiom min16 == -32768;
const unique max16:int;
axiom max16 == 32767;
const unique min32:int;
axiom min32 == -2147483648;
const unique max32:int;
axiom max32 == 2147483647;
const unique min64:int;

```

```

axiom min8 == -9223372036854775808;
const unique max64:int;
axiom max8== 9223372036854775807;
function Isint8(int) returns (bool);
axiom (forall x:int :: Isint8(x) <==> min8 <=x&&x<=max8);
function Isint16(int) returns (bool);
axiom (forall x:int :: Isint16(x) <==> min16 <=x&&x<=max16
      );
function Isint32(int) returns (bool);
axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
      );
function Isint64(int) returns (bool);
axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
      max64);

const unique TenPow16 : real;
axiom TenPow16 == 1000000000000000.0; // 10 ^ 16
const unique TenPow32 : real;
axiom TenPow32 == 10000000000000000000000000000000.0; //
      10 ^ 32
const unique TenPow64 : real;
axiom TenPow64 == 10000000000000000000000000000000000000000000.0; //
      10 ^ 64

const unique minReal16:real;
axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
const unique maxReal16:real;
axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

const unique minReal32:real;
axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
      3.402823466 E - 38
const unique maxReal32:real;
axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
      //3.402823465 E + 38

const unique minReal64:real;
axiom minReal64 == -1.7976931348623157 / (TenPow64*
      TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0) ;
      // -1.7976931348623157 E - 308
const unique maxReal64:real;
axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64*
      *TenPow64*TenPow64*TenPow32*TenPow16*1000.0); //
      1.7976931348623156 E + 308

```

```

function isReal16(real) returns (bool);
axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
       x <= maxReal16);

function isReal32(real) returns (bool);
axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
       x <= maxReal32);

function isReal64(real) returns (bool);
axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
       x <= maxReal64);

function isBool(bool) returns (bool);
axiom (forall x: bool :: isBool(x) <==> x == true || x ==
      false);

// Buffer Translation

type ClassName;
function dtype(Ref) returns (ClassName);

const unique Buffer:ClassName;
const unique Buffer.buf:Field(ArrayRef real);
const unique Buffer.front:Field int;
const unique Buffer.rear:Field int;
const unique Buffer.full:Field int;
const unique Buffer.size:int;
axiom Buffer.size==10;
const unique Buffer.value :Field real;
const unique Buffer.ovalue : Field real;

//put all the initializations of fields in construction
phase.

procedure Buffer.t0(this:Ref)
modifies Heap, ArrayHeap;
requires dtype(this) <: Buffer;
{
var oldHeap, preHeap, Heap_tmp:HeapType;
var oldArrayHeap, preArrayHeap, ArrayHeap_tmp:ArrayHeapType;
var Permission, oldPermission, prePermission:PermissionType;
var ArrayPermission, oldArrayPermission, preArrayPermission:
    ArrayPermissionType;

```

```

//initial permission
oldPermission := Permission;
havoc Permission;
assume (forall <x> r:Ref, f:Field x :: Permission [r,f] ==
        0.0);

//array initial permission
oldArrayPermission := ArrayPermission;
havoc ArrayPermission;
assume (forall <x> r:ArrayRef x, f: int :: ArrayPermission
        [r,f] == 0.0);

//claim front,rear, full
oldPermission := Permission;
havoc Permission;
assume Permission[this,Buffer.front]==1.0;
assume Permission[this,Buffer.rear]==1.0;
assume Permission[this,Buffer.full]==1.0;
assume(forall <x> r:Ref, f:Field x :: !(r==this && f==
    Buffer.front) && !(r==this && f==Buffer.rear)&&
    !(r==this&&f==Buffer.full) ==> Permission[r,f] ==
    oldPermission[r,f]);

//claim {i:{0,..size}.buf[i]}
oldArrayPermission := ArrayPermission;
havoc ArrayPermission;
assume (forall <x> r:ArrayRef x, f : int :: (r==Heap[this,
    Buffer.buf]) && (0<=f&& f<Buffer.size)==>
    ArrayPermission[r,f]==oldArrayPermission[r,f]);

//initial valuses of claimed locations
oldHeap:=Heap;
havoc Heap;
assume Heap[this,Buffer.rear] == 0;
assume Heap[this,Buffer.front]== 0;
assume Heap[this,Buffer.full]== 0;
assume (forall <x> r:Ref,f:Field x :: !(r==this && f==
    Buffer.front) && !(r==this && f==Buffer.rear) && !(r==
    this && f==Buffer.full) ==> Heap[r,f] == oldHeap[r,f])
    ;
oldArrayHeap := ArrayHeap;
havoc ArrayHeap;
assume(forall <x> r:ArrayRef x,f:int :: !((r==Heap[this,
    Buffer.buf])&& (0<=f&&f<Buffer.size))==> ArrayHeap[r,f]

```

```

]==oldArrayHeap[r,f]);
oldPermission := Permission;
oldArrayPermission := ArrayPermission;
oldHeap := Heap;
oldArrayHeap := ArrayHeap;

//While Loop
while(true)
invariant Permission[this,Buffer.front] == 1.0 &&
Permission[this,Buffer.rear] == 1.0 && Permission[this,
,Buffer.full] == 1.0 && (forall <x> r:ArrayRef x, f:
int::(r==Heap[this,Buffer.buf]))&&(0<=f &&f < Buffer.
size) ==> ArrayPermission[r,f] == 1.0);

invariant(forall<x> r:Ref, f:Field x :: !(r==this && f ==
Buffer.front)&& !(r==this && f== Buffer.rear) && !(r==
this&&f==Buffer.full)==> Permission [r,f]==
oldPermission[r,f]);

invariant(forall<x> r:ArrayRef x, f:int :: !((r==Heap[this
,Buffer.buf])&&(0<=f&&f<Buffer.size))==>
ArrayPermission[r,f]==oldArrayPermission[r,f]);

invariant 0<= Heap[this, Buffer.front] && Heap[this,
Buffer.front] < Buffer.size && (0<=Heap[this,Buffer.
rear]&& Heap[this,Buffer.rear] <Buffer.size)&& (0<
Heap[this,Buffer.full] && Heap[this,Buffer.full
]<=Buffer.size);

invariant ((Heap[this,Buffer.front]+ Heap[this,Buffer.full
]) mod Buffer.size) == Heap[this, Buffer.rear];

invariant(forall<x> r:Ref, f: Field x:: !(r==this && f ==
Buffer.front)&&! (r==this && f==Buffer.rear) && !(r==
this && f== Buffer.full) && !(r==this&&f==Buffer.
value) && !(r== this && f== Buffer.getvalue) ==> Heap [
r,f] == oldHeap[r,f]);

invariant(forall <x> r:ArrayRef x,f:int:: !((r==Heap[this ,
Buffer.buf])&&(0<=f && f<Buffer.size))==> ArrayHeap[r,
f] == oldArrayHeap[r,f]);

//while body
{
  goto deposit,fetch;
}

```

```

deposit:
assert Permission[this, Buffer.full]>0.0;
if(Heap[this, Buffer.full]<Buffer.size)
{
    prePermission := Permission;
    Permission[this, Buffer.value]:= Permission[this, Buffer.
        value]+0.5;
    if(prePermission[this, Buffer.value]==0.0)
    {
        assert Permission[this, Buffer.value]>0.0;
        havoc Heap_tmp;
        Heap[this, Buffer.value]:= Heap_tmp[this, Buffer.value];
    }
    //deposit body
    preHeap := Heap;
    preArrayHeap := ArrayHeap;
    assert Permission[this, Buffer.rear] > 0.0 && Permission
        [this, Buffer.value]>0.0;
    assert ArrayPermission[Heap[this, Buffer.buf], Heap[this,
        Buffer.rear]] == 1.0;
    assert 0<= Heap[this, Buffer.rear] && Heap[this, Buffer.
        rear]< Buffer.size;
    assert Isint32(Heap[this, Buffer.rear]);
    ArrayHeap[Heap[this, Buffer.buf], Heap[this, Buffer.rear]]
        := Heap [this, Buffer.value];
    assert Permission[this, Buffer.rear]==1.0;
    assert Isint32(Heap[this, Buffer.rear]);
    Heap[this, Buffer.rear] := (Heap[this, Buffer.rear]+1) mod
        Buffer.size;
    assert Isint32(Heap[this, Buffer.full]);
    assert Permission[this, Buffer.full] == 1.0;
    assert Isint32(Heap[this, Buffer.full]);
    Heap[this, Buffer.full] := Heap[this, Buffer.full]+1;
    assert Isint32 (Heap[this, Buffer.full]);

    //give permission
    assert Permission[this, Buffer.value] >= 0.5;
    Permission[this, Buffer.value] := Permission[this, Buffer.
        value]-0.5;
}
goto Done;
fetch:
    assert Permission[this, Buffer.full]>0.0;
    if(0< Heap[this, Buffer.full])
    {

```

```

Permission[this , Buffer.ovalue] := Permission[this ,
    Buffer.ovalue]+1.0;
havoc Heap_tmp;
Heap[this ,Buffer.ovalue] := Heap_tmp[this ,Buffer.ovalue
];
// body fetch
preHeap :=Heap;
assert Permission[this ,Buffer.ovalue] == 1.0 &&
    Permission[this ,Buffer.front]>0.0;
assert ArrayPermission[Heap[this ,Buffer.buf],Heap[this ,
    Buffer.front]]>0.0;
assert Isint32(Heap[this ,Buffer.front]);
Heap[this ,Buffer.front] := (Heap[this ,Buffer.front]+1)
    mod Buffer.size;
assert Isint32(Heap[this ,Buffer.front]);
Heap[this ,Buffer.front] := (Heap[this ,Buffer.front]+1)
    mod Buffer.size;
assert Isint32(Heap[this ,Buffer.front]);
assert Permission[this ,Buffer.full] == 1.0;
assert Isint32(Heap[this ,Buffer.full]);
Heap[this ,Buffer.full]:= Heap[this ,Buffer.full]-1;
assert Isint32(Heap[this ,Buffer.full]);
assert Permission[this ,Buffer.ovalue] == 1.0;
Permission[this ,Buffer.ovalue]:= Permission[this ,Buffer .
    ovalue]-1.0;
}
goto Done;
Done:
}
}

```

Listing A.3.2: Buffer - Code Generator Output

A.4 Permission Transfer Scenario 1

The class diagram in Figure A.5 and message sequence in Figure A.6 are visual representations of properties and internal operations of *Scenario1* class.

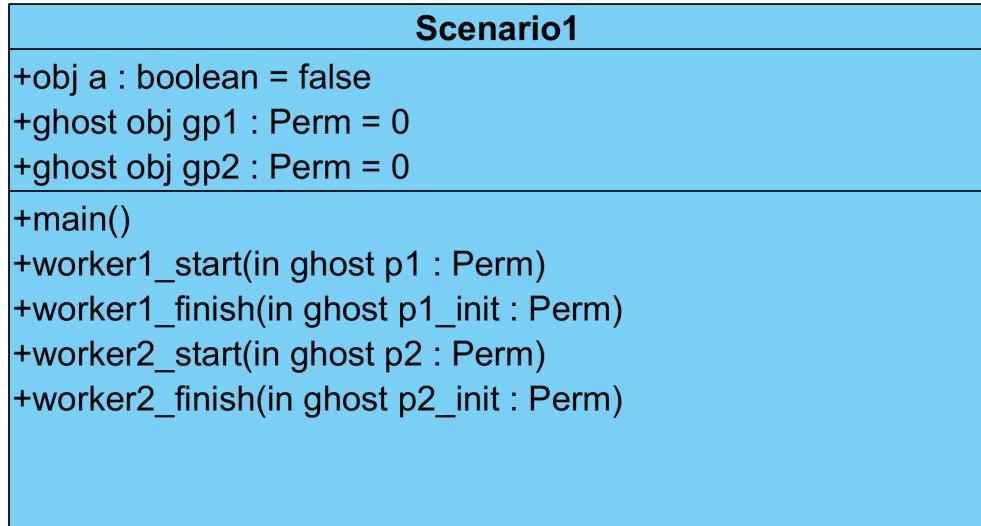


Figure A.5: Scenario 1 Class Diagram

```
(class Scenario1()
public obj a : bool := 0;
public ghost obj gp1 : Real32 := 0;
public ghost obj gp2 : Real32 := 0;
public proc main()
/* receives read permission on a and stores
 * the amount of gp1/
public proc worker1_start(ghost in p1: Real32)
  takes a@p1
  pre 0 < p1 /\ p1 /\ 1.0
  post gp1' = p1
  gives gp1@0.5
/* returns read permission equal to received
 * permission via worker1_start */
public proc worker2_start(ghost in p2_int : Real32)
  takes gp2@0.5
  pre p2_intit = gp2
  gives a@p2_init
```

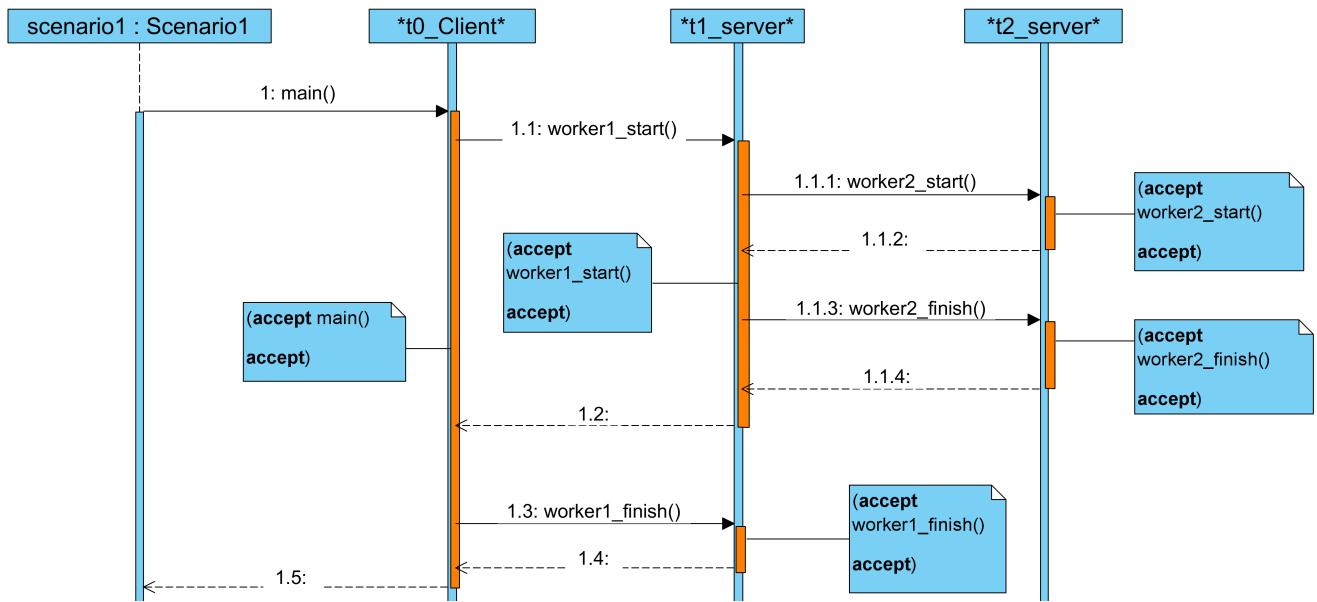


Figure A.6: Scenario 1 Message Sequence Diagram

```
//t0 has full permission on field a
/* gives read permission to t1 and finally
 * receives back it from t1 */
(thread (*t0_client*) claim a@1.0
(while true
    invariant canWrite(a)
    do
        (accept main()
            ghost obj pc1 : Real32 :=0.5
            worker1_start(pc1)
            worker1_finish(pc1)
            accept)
        while)
    thread)
(thread (*t1_server1*) claim gp1@1.0
(while true
    invariant canWrite(gp1)
    do
        (accept worker1_start(ghost in p1: Real32)
            gp1 := p1
            accept)
        ghost obj pc2 : Real32 := gp1/2
        worker2_start(pc2)
        worker2_finish(pc2))
```

```

(accept worker1_finish(ghost in p1_init : Real32)
  accept)
  while)
thread)

(thread (*t2_server2*) claim gp2@1.0
  (while true
    invariant canWrite(gp2)
    do
      (accept worker2_start(ghost in p2 : Real32)
        gp2 :=p2
        accept)
      (accept worker2_finish(ghost in p2_init :Real32)
        accept)
    while)
  thread)
class)

```

Listing A.4.1: Scenario 1 HARPO Program

```

// Prelude begin
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;

var Heap:HeapType;

type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;
var ArrayHeap:ArrayHeapType;

type Perm = real;
type PermissionType = <x> [Ref,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;

function Length<x>(Field (ArrayRef x)) returns(int);

const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;

const unique min16:int;

```



```

00000000000000000000000000000000.0; // 10 ^ 64

const unique minReal16:real;
axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
const unique maxReal16:real;
axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

const unique minReal32:real;
axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
3.402823466 E - 38
const unique maxReal32:real;
axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
//3.402823465 E + 38

const unique minReal64:real;
axiom minReal64 == -1.7976931348623157 / (TenPow64*
TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0) ;
// -1.7976931348623157 E - 308
const unique maxReal64:real;
axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64
*TenPow64*TenPow64*TenPow32*TenPow16*1000.0); //
1.7976931348623156 E + 308

function isReal16(real) returns (bool);
axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
x <= maxReal16);

function isReal32(real) returns (bool);
axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
x <= maxReal32);

function isReal64(real) returns (bool);
axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
x <= maxReal64);

function isBool(bool) returns (bool);
axiom (forall x: bool :: isBool(x) <==> x == true || x ==
false);

type ClassName;
function dtype(Ref) returns (ClassName);
// Prelude ends
const unique C : ClassName;

```

```

const unique Scenario1.a : Field bool;
const unique Scenario1.gp1 : Field Perm;
const unique Scenario1.gp2 : Field Perm;
const unique Scenario1.p1 : Field Perm;
const unique Scenario1.p2 : Field Perm;
const unique Scenario1.p1_init : Field Perm;
const unique Scenario1.p2_init : Field Perm;
const unique Scenario1.pc1 : Field Perm;
const unique Scenario1.pc2 : Field Perm;
procedure Scenario1.Constructor(This.Scenario1:Ref)
requires dtype(This.Scenario1) <: C;
modifies Heap;
modifies LockPermission;
{
var Permission : PermissionType where
(forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
var oldHeap : HeapType;
havoc Heap;
//check claims
Permission[This.Scenario1, Scenario1.a] := Permission[This
.Scenario1, Scenario1.a] + 1.0;
Permission[This.Scenario1, Scenario1.gp1] := Permission[
This.Scenario1, Scenario1.gp1] + 1.0;
Permission[This.Scenario1, Scenario1.gp2] := Permission[
This.Scenario1, Scenario1.gp2] + 1.0;
assert (forall<x> r:Ref, f : Field x :: Permission[r, f]
<= 1.0);
//initialize
Heap[This.Scenario1, Scenario1.a] := false;
Heap[This.Scenario1, Scenario1.gp1] := 0.0;
Heap[This.Scenario1, Scenario1.gp2] := 0.0;
}
procedure Scenario1.t0_client(This.Scenario1 : Ref)
modifies Heap;
requires dtype(This.Scenario1) <: C;
{
var Permission : PermissionType where
(forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
var oldHeap, preHeap, Heap_tmp : HeapType;
var p1, p1_init : Perm;
var that : Ref;
//claim
Permission[This.Scenario1, Scenario1.a] := Permission[This
.Scenario1, Scenario1.a] + 1.0;
oldHeap := Heap;

```

```

havoc Heap;
assume Heap[This.Scenario1, Scenario1.a] == false;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1.a) ==> Heap[r, f] ==
    oldHeap[r, f]);
while(true)
invariant Permission[This.Scenario1, Scenario1.a] == 1.0;
{
//thread body
Permission[This.Scenario1, Scenario1.pc1] := Permission[
    This.Scenario1, Scenario1.pc1] + 1.0;
assert Permission[This.Scenario1, Scenario1.pc1] == 1.0;
Heap[This.Scenario1, Scenario1.pc1] := 0.5;
//call worker1_start
preHeap := Heap;
assert Permission[This.Scenario1, Scenario1.pc1] > 0.0;
p1 := Heap[This.Scenario1, Scenario1.pc1];
that := This.Scenario1;
//pre-condition
assert p1 > 0.0 && p1 < 1.0;
assert Permission[that, Scenario1.a] >= p1;
Permission[that, Scenario1.a] := Permission[that,
    Scenario1.a] - p1;
//post-condition
if (Permission[that, Scenario1.gp1] == 0.0)
{
havoc Heap_tmp;
Heap[that, Scenario1.gp1] := Heap_tmp[that, Scenario1.gp1
    ];
}
Permission[that, Scenario1.gp1] := Permission[that,
    Scenario1.gp1] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[that, Scenario1.gp1] == p1;
assume (forall <x> r:Ref, f : Field x :: !(r==that && f ==
    Scenario1.gp1) ==> Heap[r, f] == oldHeap[r, f]);
//call worker1_finish
preHeap := Heap;
that := This.Scenario1;
assert Permission[This.Scenario1, Scenario1.pc1] > 0.0;
p1_init := Heap[This.Scenario1, Scenario1.pc1];
//pre-condition
assert p1_init == Heap[that, Scenario1.gp1];
assert Permission[that, Scenario1.gp1] >= 0.5;

```

```

Permission[that, Scenario1_gp1] := Permission[that,
    Scenario1_gp1] - 0.5;
//post-condition
if (Permission[that, Scenario1.a] == 0.0)
{
    havoc Heap_tmp;
Heap[that, Scenario1.a] := Heap_tmp[that, Scenario1.a];
}
Permission[that, Scenario1.a] := Permission[that,
    Scenario1.a] + p1_init;
//test A
assert Permission[This.Scenario1, Scenario1.a] == 1.0;
assert Permission[This.Scenario1, Scenario1_gp1] == 0.0;
assert Permission[This.Scenario1, Scenario1_gp2] == 0.0;
//lose local permission
assert Permission[This.Scenario1, Scenario1_pc1] == 1.0;
Permission[This.Scenario1, Scenario1_pc1] := Permission[
    This.Scenario1, Scenario1_pc1] - 1.0;
}
}
procedure Scenario1.t1_server1(This.Scenario1 : Ref)
modifies Heap;
requires dtype(This.Scenario1) <: C;
{
var Permission : PermissionType where
(forall <x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
var oldHeap, Heap_tmp, preHeap : HeapType;
var p2, p2_init : Perm;
var that : Ref;
//claim
Permission[This.Scenario1, Scenario1_gp1] := Permission[
    This.Scenario1, Scenario1_gp1] + 1.0;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario1, Scenario1_gp1] == 0.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1_gp1) ==> Heap[r,f] ==
    oldHeap[r, f]);
while(true)
invariant Permission[This.Scenario1, Scenario1_gp1] ==
    1.0;
{
//worker1_start
goto worker1_start;
worker1_start:
}
}

```

```

//pre-condition
if (Permission[This.Scenario1, Scenario1.p1] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.p1] := Heap_tmp[This.
    Scenario1, Scenario1.p1];
}
Permission[This.Scenario1, Scenario1.p1] := Permission[
    This.Scenario1, Scenario1.p1] + 0.5;
oldHeap := Heap;
havoc Heap;

assume Heap[This.Scenario1, Scenario1.p1] > 0.0 && Heap[
    This.Scenario1, Scenario1.p1] < 1.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1.p1) ==> Heap[r, f] ==
    oldHeap[r, f]);
if (Permission[This.Scenario1, Scenario1.a] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.a] := Heap_tmp[This.
    Scenario1, Scenario1.a];
}
Permission[This.Scenario1, Scenario1.a] := Permission[This
    .Scenario1, Scenario1.a] + Heap[This.Scenario1,
    Scenario1.p1];
//body
preHeap := Heap;
assert Permission[This.Scenario1, Scenario1.p1] > 0.0;
assert Permission[This.Scenario1, Scenario1.gp1] == 1.0;
Heap[This.Scenario1, Scenario1.gp1] := Heap[This.Scenario1
    , Scenario1.p1];
//post-condition
assert Heap[This.Scenario1, Scenario1.gp1] == preHeap[This
    .Scenario1, Scenario1.p1];
Permission[This.Scenario1, Scenario1.gp1] := Permission[
    This.Scenario1, Scenario1.gp1] - 0.5;
Permission[This.Scenario1, Scenario1.p1] := Permission[
    This.Scenario1, Scenario1.p1] - 0.5;
goto Done_worker1_start;
Done_worker1_start:
//thread code
Permission[This.Scenario1, Scenario1.pc2] := Permission[
    This.Scenario1, Scenario1.pc2] + 1.0;
assert Permission[This.Scenario1, Scenario1.pc2] == 1.0;

```

```

assert Permission[This.Scenario1, Scenario1.gp1] > 0.0;
Heap[This.Scenario1, Scenario1.pc2] := Heap[This.Scenario1
    , Scenario1.gp1] / 2;
//call worker2_start
oldHeap := Heap;
assert Permission[This.Scenario1, Scenario1.pc2] > 0.0;
p2 := Heap[This.Scenario1, Scenario1.pc2];
that := This.Scenario1;
//pre-condition
assert p2 > 0.0 && p2 < 1.0;
assert Permission[that, Scenario1.a] >= p2;
Permission[that, Scenario1.a] := Permission[that,
    Scenario1.a] - p2;
//post-condition
if (Permission[that, Scenario1.gp2] == 0.0)
{
havoc Heap_tmp;
Heap[that, Scenario1.gp2] := Heap_tmp[that, Scenario1.gp2
    ];
}
Permission[that, Scenario1.gp2] := Permission[that,
    Scenario1.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[that, Scenario1.gp2] == p2;
assume (forall <x> r:Ref, f : Field x :: !(r==that && f ==
    Scenario1.gp2) ==> Heap[r, f] == oldHeap[r, f]);
//call worker2_finish
oldHeap := Heap;
assert Permission[This.Scenario1, Scenario1.pc2] > 0.0;
p2_init := Heap[This.Scenario1, Scenario1.pc2];
that := This.Scenario1;
//pre-condition
assert p2_init == Heap[that, Scenario1.gp2];
Permission[that, Scenario1.gp2] := Permission[that,
    Scenario1.gp2] - 0.5;
//post-condition
if (Permission[that, Scenario1.a] == 0.0)
{
havoc Heap_tmp;
Heap[that, Scenario1.a] := Heap_tmp[that, Scenario1.a];
}
Permission[that, Scenario1.a] := Permission[that,
    Scenario1.a] + p2_init;
//worker1_finish

```

```

goto worker1_finish;
worker1_finish:
//pre-condition
if (Permission[This.Scenario1, Scenario1.p1_init] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.p1_init] := Heap_tmp[This.
    Scenario1, Scenario1.p1_init];
}
Permission[This.Scenario1, Scenario1.p1_init] :=
    Permission[This.Scenario1, Scenario1.p1_init] + 0.5;
if (Permission[This.Scenario1, Scenario1.gp1] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.gp1] := Heap_tmp[This.
    Scenario1, Scenario1.gp1];
}
Permission[This.Scenario1, Scenario1.gp1] := Permission[
    This.Scenario1, Scenario1.gp1] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario1, Scenario1.p1_init] == Heap[
    This.Scenario1, Scenario1.gp1];
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1.p1_init) ==> Heap[r, f] ==
    oldHeap[r, f]);
//body
preHeap := Heap;
//post
Permission[This.Scenario1, Scenario1.a] := Permission[This
    .Scenario1, Scenario1.a] - Heap[This.Scenario1,
    Scenario1.p1_init];
Permission[This.Scenario1, Scenario1.p1_init] :=
    Permission[This.Scenario1, Scenario1.p1_init] - 0.5;
goto Done_worker1_finish;
Done_worker1_finish:
//test B
assert Permission[This.Scenario1, Scenario1.a] == 0.0;
assert Permission[This.Scenario1, Scenario1.gp1] == 1.0;
assert Permission[This.Scenario1, Scenario1.gp2] == 0.0;
//lose permission on locals
assert Permission[This.Scenario1, Scenario1.pc2] == 1.0;
Permission[This.Scenario1, Scenario1.pc2] := Permission[
    This.Scenario1, Scenario1.pc2] - 1.0;
}

```

```

}

procedure Scenario1.t2_server2(This.Scenario1 : Ref)
modifies Heap;
requires dtype(This.Scenario1) <: C;
{
var Permission : PermissionType where
(forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
var oldHeap, Heap_tmp, preHeap : HeapType;
var oldPermission : PermissionType;
//claim gp2@1.0
Permission[This.Scenario1, Scenario1(gp2) := Permission[
    This.Scenario1, Scenario1(gp2)] + 1.0;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario1, Scenario1(gp2)] == 0.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1(gp2)) ==> Heap[r, f] ==
    oldHeap[r, f]);
while(true)
invariant Permission[This.Scenario1, Scenario1(gp2)] ==
    1.0;
{
//worker2_start
goto worker2_start;
worker2_start:
//pre
if (Permission[This.Scenario1, Scenario1.p2] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.p2] := Heap_tmp[This.
    Scenario1, Scenario1.p2];
}

Permission[This.Scenario1, Scenario1.p2] := Permission[
    This.Scenario1, Scenario1.p2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume 0.0 < Heap[This.Scenario1, Scenario1.p2] && Heap[
    This.Scenario1, Scenario1.p2] < 1.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1.p2)) ==> Heap[r, f] ==
    oldHeap[r, f]);
if (Permission[This.Scenario1, Scenario1.a] == 0.0)
{
havoc Heap_tmp;
}

```

```

Heap[This.Scenario1, Scenario1.a] := Heap_tmp[This.
    Scenario1, Scenario1.a];
}
Permission[This.Scenario1, Scenario1.a] := Permission[This
    .Scenario1, Scenario1.a] + Heap[This.Scenario1,
    Scenario1.p2];
//body
preHeap := Heap;
assert Permission[This.Scenario1, Scenario1.p2] > 0.0;
assert Permission[This.Scenario1, Scenario1.gp2] == 1.0;
Heap[This.Scenario1, Scenario1.gp2] := Heap[This.Scenario1
    , Scenario1.p2];
//post-condition
assert Heap[This.Scenario1, Scenario1.gp2] == preHeap[This
    .Scenario1, Scenario1.p2];
Permission[This.Scenario1, Scenario1.gp2] := Permission[
    This.Scenario1, Scenario1.gp2] - 0.5;
Permission[This.Scenario1, Scenario1.p2] := Permission[
    This.Scenario1, Scenario1.p2] - 0.5;
goto Done_worker2_start;
Done_worker2_start:
//worker2_finish
goto worker2_finish;
worker2_finish:
//pre-condition
if (Permission[This.Scenario1, Scenario1.p2_init] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.p2_init] := Heap_tmp[This.
    Scenario1, Scenario1.p2_init];
}
Permission[This.Scenario1, Scenario1.p2_init] :=
    Permission[This.Scenario1, Scenario1.p2_init] + 0.5;
if (Permission[This.Scenario1, Scenario1.gp2] == 0.0)
{
havoc Heap_tmp;
Heap[This.Scenario1, Scenario1.gp2] := Heap_tmp[This.
    Scenario1, Scenario1.gp2];
}
Permission[This.Scenario1, Scenario1.gp2] := Permission[
    This.Scenario1, Scenario1.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario1, Scenario1.p2_init] == Heap[
    This.Scenario1, Scenario1.gp2];

```

```

assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario1 && f == Scenario1.p2_init) ==> Heap[r, f] ==
    oldHeap[r, f]);
//body
preHeap := Heap;
//post
assert Permission[This.Scenario1, Scenario1.a] >= Heap[
    This.Scenario1, Scenario1.p2_init];
Permission[This.Scenario1, Scenario1.a] := Permission[This
    .Scenario1, Scenario1.a] - Heap[This.Scenario1,
    Scenario1.p2_init];
assert Permission[This.Scenario1, Scenario1.p2_init] >=
    0.5;
Permission[This.Scenario1, Scenario1.p2_init] :=
    Permission[This.Scenario1, Scenario1.p2_init] - 0.5;
goto Done_worker2_finish;
Done_worker2_finish:
//test C
assert Permission[This.Scenario1, Scenario1.a] == 0.0;
assert Permission[This.Scenario1, Scenario1.gp1] == 0.0;
assert Permission[This.Scenario1, Scenario1.gp2] == 1.0;
}
}

```

Listing A.4.2: Scenario 1 Code Generator Output

A.5 Permission Transfer Scenario 2

The class diagram in Figure A.7 and message sequence in Figure A.8 are visual representations of properties and internal operations of *Scenario2* class.

Scenario2
-obj a : boolean = false
+ghost obj gp1 : Perm = 0
-ghost obj gp2 : Perm = 0
+main()
+worker1_finish(in ghost p1_init : Perm)
+worker1_start(in ghost p1 : Perm)
+worker2_start(in ghost p2 : Perm)
+worker2_finish(in ghost p2_init : Perm, out ghost po2 : Perm)

Figure A.7: Scenario 2 Class Diagram

```
(class Scenario2()
public proc main()
/* Receives the permission on field a and stores
 * the amount of gp1 gives read permission to t2 */
public proc worker1_start(ghost in p1: Real64)
takes a@p1
pre 0 < p1 /\ p1 < 1.0
post gp1' = p1
post gp2' = p1/2
gives gp1@0.5 , gp2@0.5
/* returns read permission to caller
 * the retuned permission is the half of the permission
 * in
 * received via worker1_start */

public proc worker1_finish(ghost in p1_init : Real64,
                           out po1: Real64)
takes gp1@0.5
pre p1_init = gp1
post po1' = p1_init/2
```

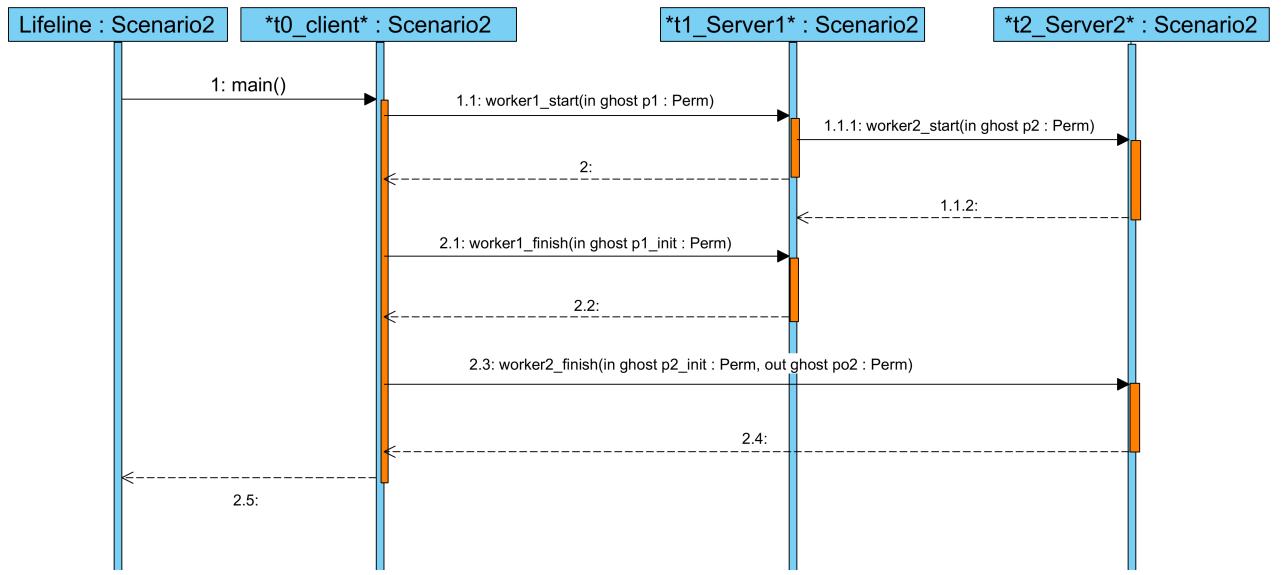


Figure A.8: Scenario 2 Message Sequence Diagram

```

gives a@po1
/* receives read permission on field a and
 * stores the amount in gp2 */
public proc worker2_start(ghost in p2:Real64)
    takes a@p2
    pre 0<p2 and p2<1.0
    post gp2'= p2
    gives gp2@0.5
    /* return read permission to caller
     * the returned permission is equal to the
     * permission it received via worker2_start */

public proc worker2_finish(ghost in p2_init: Real64,
                           ghost out po2: Real64)
    takes gp2@0.5
    pre p2_init = gp2
    post po2' = p2_init
    gives a@po2

public obj a: Bool := false
public ghost obj gp1 : Real64 := 0
public ghost obj gp2 : Real64 := 0

(thread (*t0_client*) claim a@1.0
  (while true

```

```

do
(accept main()
ghost obj p : Real64 :=0
ghost obj pc1: Real64 :=0
ghost obj pr1 : Real64 :=0
ghost obj pr2 : Real64 :=0
p:=1
pc1 := p/2
p := p-pc1
worker1_start(pc1)
worker1_finish(pc1,pr1)
p := p+pr1;
worker2_finish(gp2,pr2)
p :=p+pr2
assert p = 1
accept)
while)
thread)

(thread (*t1_server1*) claim gp1@1.0
}while true
do
ghost obj pc2 : Real64 := 0;
(accept worker1_start(ghost in p1: Real64)
gp1 := p1
pc2 :=gp1/2
worker2_start(pc2)
accept)
(accept worker1_finish(ghost in p1_init : Real64,
ghost out po1: Real64)
accept)
while)
thread)

(thread (*t2_server2*) claim gp2@1
}while true
do
(accept worker2_start (ghost in p2: Real64)
gp2 := p2
accept)
(accept worker2_finish(ghost in p2_init: Real64,
ghost out po2 : Real64)
po2 := p2_init
accept)
while)

```

```
thread)
class)
```

Listing A.5.1: Scenario 2

```
// Prelude begin
type Ref;
type Field x;
type HeapType = <x> [Ref,Field x]x;

var Heap:HeapType;

type ArrayRef x;
type ArrayHeapType = <x> [ArrayRef x, int]x;
var ArrayHeap:ArrayHeapType;

type Perm = real;
type PermissionType = <x> [Ref,Field x]Perm;
type ArrayPermissionType = <x>[ArrayRef x, int] Perm;
var LockPermission: PermissionType;
var ArrayLockPermission: ArrayPermissionType;

function Length<x>(Field (ArrayRef x)) returns (int);

const unique min8:int;
axiom min8 == -128;
const unique max8:int;
axiom max8==127;

const unique min16:int;
axiom min16 == -32768;
const unique max16:int;
axiom max16 == 32767;

const unique min32:int;
axiom min32 == -2147483648;
const unique max32:int;
axiom max32 == 2147483647;

const unique min64:int;
axiom min64 == -9223372036854775808;
const unique max64:int;
axiom max64== 9223372036854775807;

function Isint8(int) returns (bool);
axiom (forall x:int :: Isint8(x) <=> min8 <=x&&x<=max8);
```

```

function Isint16(int) returns (bool);
axiom (forall x:int :: Isint16(x) <==> min16 <=x&&x<=max16
);

function Isint32(int) returns (bool);
axiom (forall x:int :: Isint32(x) <==> min32 <=x&&x<=max32
);

function Isint64(int) returns (bool);
axiom (forall x:int :: Isint64(x) <==> min64 <= x && x <=
max64);

const unique minPer : Perm;
axiom minPer == 0.0;
const unique maxPer : Perm;
axiom maxPer == 1.0;

function IsValidPermission(Perm) returns (bool);
axiom (forall x: Perm:: IsValidPermission(x) <==> minPer
<= x && x <= maxPer);

const unique TenPow16 : real;
axiom TenPow16 == 1000000000000000.0; // 10 ^ 16
const unique TenPow32 : real;
axiom TenPow32 == 10000000000000000000000000000000.0; //
10 ^ 32
const unique TenPow64 : real;
axiom TenPow64 == 10000000000000000000000000000000000000000000.0; // 10 ^ 64

const unique minReal16:real;
axiom minReal16 == -6.10/10000.0; // 6.10 E - 5
const unique maxReal16:real;
axiom maxReal16 == 6.55*10000.0; // 6.55 E + 4

const unique minReal32:real;
axiom minReal32 == -3.402823466 / (TenPow32*10000.0) ; //
3.402823466 E - 38
const unique maxReal32:real;
axiom maxReal32 == 3.402823465 * (TenPow32*10000.0) ;
//3.402823465 E + 38

const unique minReal64:real;
axiom minReal64 == -1.7976931348623157 / (TenPow64*

```

```

TenPow64*TenPow64*TenPow64*TenPow32*TenPow16*1000.0) ;
// -1.7976931348623157 E - 308
const unique maxReal64:real;
axiom maxReal64 == 1.7976931348623156 * (TenPow64*TenPow64
*TenPow64*TenPow64*TenPow32*TenPow16*1000.0); // 
1.7976931348623156 E + 308

function isReal16(real) returns (bool);
axiom (forall x:real :: isReal16(x) <==> minReal16 <= x &&
x <= maxReal16);

function isReal32(real) returns (bool);
axiom (forall x:real :: isReal32(x) <==> minReal32 <= x &&
x <= maxReal32);

function isReal64(real) returns (bool);
axiom (forall x:real :: isReal64(x) <==> minReal64 <= x &&
x <= maxReal64);

function isBool(bool) returns (bool);
axiom (forall x: bool :: isBool(x) <==> x == true || x ==
false);

type ClassName;
function dtype(Ref) returns (ClassName);
// Prelude ends
const unique Scenario2 : ClassName;
const unique Scenario2.a : Field bool;
const unique Scenario2.gp1 : Field Perm;
const unique Scenario2.gp2 : Field Perm;
const unique Scenario2.p1 : Field Perm;
const unique Scenario2.p2 : Field Perm;
const unique Scenario2.p1_init : Field Perm;
const unique Scenario2.p2_init : Field Perm;
const unique Scenario2.po1 : Field Perm;
const unique Scenario2.po2 : Field Perm;
const unique Scenario2.p : Field Perm;
const unique Scenario2.pc1 : Field Perm;
const unique Scenario2.pc2 : Field Perm;
procedure Scenario2.Constructor(This.Scenario2:Ref)
requires dtype(This.Scenario2) <: Scenario2;
modifies Heap;
modifies LockPermission;
{

```

```

var Permission : PermissionType where
(forall<x> r:Ref, f : Field x :: Permission[r, f] ==
0.0);
var oldHeap : HeapType;
havoc Heap;
//check claims
Permission[This.Scenario2, Scenario2.a] := Permission[
    This.Scenario2, Scenario2.a] + 1.0;
Permission[This.Scenario2, Scenario2_gp1] := Permission[
    This.Scenario2, Scenario2_gp1] + 1.0;
Permission[This.Scenario2, Scenario2_gp2] := Permission[
    This.Scenario2, Scenario2_gp2] + 1.0;
assert (forall<x> r:Ref, f : Field x :: Permission[r, f]
    <= 1.0);
//initialize
Heap[This.Scenario2, Scenario2.a] := false;
Heap[This.Scenario2, Scenario2_gp1] := 0.0;
Heap[This.Scenario2, Scenario2_gp2] := 0.0;
}
procedure Scenario2.t0_client(This.Scenario2 : Ref)
modifies Heap;
requires dtype(This.Scenario2) <: Scenario2;
{
    var Permission : PermissionType where
(forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0)
    ;
var oldHeap, preHeap, preHeap0, Heap_tmp : HeapType;
var that : Ref;
var p1, p1_init, po1 : Perm;
var p2, p2_init, po2 : Perm;
//claim
Permission[This.Scenario2, Scenario2.a] := Permission[
    This.Scenario2, Scenario2.a] + 1.0;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario2, Scenario2.a] == false;
assume (forall<x> r:Ref, f : Field x :: !(r==This.
    Scenario2 && f == Scenario2.a)) ==> Heap[r, f] ==
oldHeap[r, f];
while(true)
    invariant Permission[This.Scenario2, Scenario2.a] ==
        1.0;
{
    //main
    preHeap0 := Heap;
}

```

```

//permission on local variables
Permission[This.Scenario2, Scenario2.p] := Permission[
    This.Scenario2, Scenario2.p] + 1.0;
Heap[This.Scenario2, Scenario2.p] := 0.0;
Permission[This.Scenario2, Scenario2.pc1] := Permission[
    This.Scenario2, Scenario2.pc1] + 1.0;
Heap[This.Scenario2, Scenario2.pc1] := 0.0;
Permission[This.Scenario2, Scenario2.po1] := Permission[
    This.Scenario2, Scenario2.po1] + 1.0;
Heap[This.Scenario2, Scenario2.po1] := 0.0;
Permission[This.Scenario2, Scenario2.po2] := Permission[
    This.Scenario2, Scenario2.po2] + 1.0;
Heap[This.Scenario2, Scenario2.po2] := 0.0;
//code
assert Permission[This.Scenario2, Scenario2.p] == 1.0;
Heap[This.Scenario2, Scenario2.p] := 1.0;
assert Permission[This.Scenario2, Scenario2.pc1] == 1.0
    && Permission[This.Scenario2, Scenario2.p] > 0.0;
Heap[This.Scenario2, Scenario2.pc1] := Heap[This.
    Scenario2, Scenario2.p] / 2;
assert Permission[This.Scenario2, Scenario2.p] == 1.0
    && Permission[This.Scenario2, Scenario2.pc1] >
    0.0;
Heap[This.Scenario2, Scenario2.p] := Heap[This.
    Scenario2, Scenario2.p] - Heap[This.Scenario2,
    Scenario2.pc1];
//call worker1_start
preHeap := Heap;
that := This.Scenario2;
assert Permission[This.Scenario2, Scenario2.pc1] > 0.0;
p1 := Heap[This.Scenario2, Scenario2.pc1];
//pre-condition
assert p1 > 0.0 && p1 < 1.0;
assert Permission[that, Scenario2.a] >= p1;
Permission[This.Scenario2, Scenario2.a] := Permission[
    that, Scenario2.a] - p1;
//post-condition
if (Permission[that, Scenario2.gp1] == 0.0)
{
    havoc Heap_tmp;
    Heap[that, Scenario2.gp1] := Heap_tmp[that, Scenario2.
        gp1];
}
Permission[that, Scenario2.gp1] := Permission[that,
    Scenario2.gp1] + 0.5;

```

```

if (Permission[that, Scenario2.gp2] == 0.0)
{
    havoc Heap_tmp;
    Heap[that, Scenario2.gp2] := Heap_tmp[that, Scenario2.gp2];
}
Permission[that, Scenario2.gp2] := Permission[that,
Scenario2.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[that, Scenario2.gp1] == p1;
assume Heap[that, Scenario2.gp2] == p1 / 2;
assume (forall<x> r:Ref, f : Field x :: !(r==that && f
== Scenario2.gp1)      || (r==that && f == Scenario2.
gp1)) ==>
Heap[r, f] == oldHeap[r, f]);
//call worker1_finish
preHeap := Heap;
that := This.Scenario2;
assert Permission[This.Scenario2, Scenario2.pc1] > 0.0;
p1_init := Heap[This.Scenario2, Scenario2.pc1];
//pre-condition
assert p1_init == Heap[that, Scenario2.gp1];
assert Permission[that, Scenario2.gp1] >= 0.5;
Permission[that, Scenario2.gp1] := Permission[that,
Scenario2.gp1] - 0.5;
//post-condition
havoc po1;
assume po1 == p1_init / 2;
if (Permission[that, Scenario2.a] == 0.0)
{
    havoc Heap_tmp;
    Heap[that, Scenario2.a] := Heap_tmp[that, Scenario2.a];
}
Permission[that, Scenario2.a] := Permission[that,
Scenario2.a] + po1;
assert Permission[This.Scenario2, Scenario2.po1] ==
1.0;
Heap[This.Scenario2, Scenario2.po1] := po1;
assert Permission[This.Scenario2, Scenario2.po1] > 0.0
&& Permission[This.Scenario2, Scenario2.p] == 1.0;
Heap[This.Scenario2, Scenario2.p] := Heap[This.
Scenario2, Scenario2.p]      + Heap[This.Scenario2,
Scenario2.po1];

```

```

//call worker2_finish
preHeap := Heap;
that := This.Scenario2;
assert Permission[This.Scenario2, Scenario2_gp2] > 0.0;
assert Permission[This.Scenario2, Scenario2_po2] ==
1.0;
p2_init := Heap[This.Scenario2, Scenario2_gp2];
//pre-condition
assert p2_init == Heap[that, Scenario2_gp2];
assert Permission[that, Scenario2_gp2] >= 0.5;
Permission[that, Scenario2_gp2] := Permission[that,
Scenario2_gp2] - 0.5;
//post-condition
havoc po2;
assume po2 == p2_init;
if (Permission[that, Scenario2.a] == 0.0)
{
    havoc Heap_tmp;
    Heap[that, Scenario2.a] := Heap_tmp[This.Scenario2,
Scenario2.a];
}
Permission[that, Scenario2.a] := Permission[that,
Scenario2.a] + po2; assert Permission[This.
Scenario2, Scenario2_po2] == 1.0;
Heap[This.Scenario2, Scenario2_po2] := po2;
//test A
assert Permission[This.Scenario2, Scenario2.a] == 1.0;
assert Permission[This.Scenario2, Scenario2_gp1] ==
0.0;
assert Permission[This.Scenario2, Scenario2_gp2] ==
0.0;
//lose permission on local variables
assert Permission[This.Scenario2, Scenario2_p] == 1.0;
Permission[This.Scenario2, Scenario2_p] := Permission[
This.Scenario2, Scenario2_p] - 1.0;
assert Permission[This.Scenario2, Scenario2_pc1] ==
1.0;
Permission[This.Scenario2, Scenario2_pc1] := Permission
[This.Scenario2, Scenario2_pc1] - 1.0;
assert Permission[This.Scenario2, Scenario2_po1] ==
1.0;
Permission[This.Scenario2, Scenario2_po1] := Permission
[This.Scenario2, Scenario2_po1] - 1.0;
assert Permission[This.Scenario2, Scenario2_po2] ==
1.0;

```

```

    Permission[This.Scenario2, Scenario2.po2] := Permission
        [This.Scenario2, Scenario2.po2] - 1.0;
    }
}

procedure Scenario2.t1_server1(This.Scenario2 : Ref)
modifies Heap;
requires dtype(This.Scenario2) <: Scenario2;
{
    var Permission : PermissionType where
        (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0)
        ;
    var oldHeap, oldHeap2, preHeap, Heap_tmp : HeapType;
    var that : Ref;
    var p2, p2_init : Perm;
    //claim
    Permission[This.Scenario2, Scenario2.gp1] := Permission[
        This.Scenario2, Scenario2.gp1] + 1.0;
    oldHeap := Heap;
    havoc Heap;
    assume Heap[This.Scenario2, Scenario2.gp1] == 0.0;
    assume (forall <x> r:Ref, f : Field x :: !(r==This.
        Scenario2 && f == Scenario2.gp1) ==> Heap[r, f] ==
        oldHeap[r, f]);
    while (true)
        invariant Permission[This.Scenario2, Scenario2.gp1] ==
            1.0;
    {
        //worker1_start
        goto worker1_start;
        worker1_start:
        //pre-condition
        if (Permission[This.Scenario2, Scenario2.p1] == 0.0)
        {
            havoc Heap_tmp;
            Heap[This.Scenario2, Scenario2.p1] := Heap_tmp[This.
                Scenario2, Scenario2.p1];
        }
        Permission[This.Scenario2, Scenario2.p1] := Permission[
            This.Scenario2, Scenario2.p1] + 0.5;
        oldHeap := Heap;
        havoc Heap;
        assume Heap[This.Scenario2, Scenario2.p1] > 0.0 && Heap
            [This.Scenario2, Scenario2.p1] < 1.0;
        assume (forall<x> r:Ref, f : Field x :: !(r==This.
            Scenario2 && f == Scenario2.p1)) ==> Heap[r, f] ==

```

```

        oldHeap[r, f]);
if (Permission[This.Scenario2, Scenario2.a] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2.a] := Heap_tmp[This.
    Scenario2, Scenario2.a];
}
assert Permission[This.Scenario2, Scenario2.p1] > 0.0;
Permission[This.Scenario2, Scenario2.a] := Permission[
    This.Scenario2, Scenario2.a] + Heap[This.Scenario2
    , Scenario2.p1];
//body
preHeap := Heap;
assert Permission[This.Scenario2, Scenario2.p1] > 0.0;
assert Permission[This.Scenario2, Scenario2.gp1] ==
1.0;
Heap[This.Scenario2, Scenario2.gp1] := Heap[This.
    Scenario2, Scenario2.p1];
Permission[This.Scenario2, Scenario2.pc2] := Permission
    [This.Scenario2, Scenario2.pc2] + 1.0;
assert Permission[This.Scenario2, Scenario2.pc2] ==
1.0;
assert Permission[This.Scenario2, Scenario2.gp1] > 0.0;
Heap[This.Scenario2, Scenario2.pc2] := Heap[This.
    Scenario2, Scenario2.gp1] / 2;
//call worker2_start
oldHeap2 := Heap;
that := This.Scenario2;
assert Permission[This.Scenario2, Scenario2.pc2] > 0.0;
p2 := Heap[This.Scenario2, Scenario2.pc2];
//pre-condition call worker2_start
assert p2 > 0.0 && p2 < 1.0;
assert Permission[that, Scenario2.a] >= p2;
Permission[that, Scenario2.a] := Permission[that,
    Scenario2.a] - p2;
//post-condition call worker2_start
if (Permission[This.Scenario2, Scenario2.gp2] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2.gp2] := Heap_tmp[This.
    Scenario2, Scenario2.gp2];
}
Permission[This.Scenario2, Scenario2.gp2] := Permission
    [that, Scenario2.gp2] + 0.5;
oldHeap := Heap;

```

```

havoc Heap;
assume Heap[that, Scenario2_gp2] == p2;
assume (forall<x> r:Ref, f : Field x :: (!(r==that && f
== Scenario2_gp2)) ==> Heap[r, f] == oldHeap[r, f
]);
//post-condition worker1_start
assert Heap[This.Scenario2, Scenario2_gp1] == Heap[This
.Scenario2, Scenario2_p1];
assert Heap[This.Scenario2, Scenario2_gp2] == Heap[This
.Scenario2, Scenario2_p1]/2;
Permission[This.Scenario2, Scenario2_gp2] := Permission
[This.Scenario2, Scenario2_gp2] - 0.5;
Permission[This.Scenario2, Scenario2_gp1] := Permission
[This.Scenario2, Scenario2_gp1] - 0.5;
Permission[This.Scenario2, Scenario2_p1] := Permission[
This.Scenario2, Scenario2_p1] - 0.5;
goto Done_worker1_start;
Done_worker1_start:
//worker1_finish
goto worker1_finish;
worker1_finish:
//pre-condition
if (Permission[This.Scenario2, Scenario2_p1_init] ==
0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2_p1_init] := Heap_tmp[This
.Scenario2, Scenario2_p1_init];
}
Permission[This.Scenario2, Scenario2_p1_init] :=
Permission[This.Scenario2, Scenario2_p1_init] +
0.5;
if (Permission[This.Scenario2, Scenario2_p01] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2_p01] := Heap_tmp[This
.Scenario2, Scenario2_p01];
}
Permission[This.Scenario2, Scenario2_p01] := Permission
[This.Scenario2, Scenario2_p01] + 1.0;
if (Permission[This.Scenario2, Scenario2_gp1] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2_gp1] := Heap_tmp[This
.Scenario2, Scenario2_gp1];
}

```

```

}

Permission[This.Scenario2, Scenario2.gp1] := Permission
  [This.Scenario2, Scenario2.gp1] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario2, Scenario2.p1_init] == Heap[
  This.Scenario2, Scenario2.gp1];
assume (forall<x> r:Ref, f : Field x :: !(r==This.
  Scenario2 && f == Scenario2.p1_init)) ==> Heap[r,
  f] == oldHeap[r, f]);
//body
preHeap := Heap;
assert Permission[This.Scenario2, Scenario2.po1] == 1.0
  && Permission[This.Scenario2, Scenario2.p1_init] >
  0.0;
Heap[This.Scenario2, Scenario2.po1] := Heap[This.
  Scenario2, Scenario2.gp1] / 2;
//post-condition
assert Heap[This.Scenario2, Scenario2.po1] == preHeap[
  This.Scenario2, Scenario2.p1_init] / 2;
assert Permission[This.Scenario2, Scenario2.a] >= Heap[
  This.Scenario2, Scenario2.po1];
Permission[This.Scenario2, Scenario2.a] := Permission[
  This.Scenario2, Scenario2.a] - Heap[This.Scenario2,
  Scenario2.po1];
assert Permission[This.Scenario2, Scenario2.po1] == 1.0;
Permission[This.Scenario2, Scenario2.po1] := Permission
  [This.Scenario2, Scenario2.po1] - 1.0;
assert Permission[This.Scenario2, Scenario2.p1_init] >=
  0.5;
Permission[This.Scenario2, Scenario2.p1_init] :=
  Permission[This.Scenario2, Scenario2.p1_init] - 0.5;
goto Done_worker1_finish;
Done_worker1_finish:
assert Permission[This.Scenario2, Scenario2.a] == 0.0;
assert Permission[This.Scenario2, Scenario2.gp1] == 1.0;
assert Permission[This.Scenario2, Scenario2.gp2] == 0.0;
}
}

procedure Scenario2.t2_server2(This.Scenario2 : Ref)
modifies Heap;
requires dtype(This.Scenario2) <: Scenario2;
{
  var Permission : PermissionType where
  (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0)
}

```

```

;
var oldHeap, Heap_tmp, preHeap : HeapType;
//claim
Permission[This.Scenario2, Scenario2_gp2] := Permission[
    This.Scenario2, Scenario2_gp2] + 1.0;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario2, Scenario2_gp2] == 0.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario2 && f == Scenario2_gp2) ==> Heap[r, f] ==
    oldHeap[r, f]);
while(true)
    invariant Permission[This.Scenario2, Scenario2_gp2] ==
        1.0;
{
//worker2_start
goto worker2_start;
worker2_start:
//pre-condition
if (Permission[This.Scenario2, Scenario2_p2] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2_p2] := Heap_tmp[This.
        Scenario2, Scenario2_p2];
}
Permission[This.Scenario2, Scenario2_p2] := Permission[
    This.Scenario2, Scenario2_p2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario2, Scenario2_p2] > 0.0 && Heap[
    This.Scenario2, Scenario2_p2] < 1.0;
assume (forall <x> r:Ref, f : Field x :: !(r==This.
    Scenario2 && f == Scenario2_p2)) ==> Heap[r, f] ==
    oldHeap[r, f]);
if (Permission[This.Scenario2, Scenario2_a] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2_a] := Heap_tmp[This.
        Scenario2, Scenario2_a];
}
assert Permission[This.Scenario2, Scenario2_p2] > 0.0;
Permission[This.Scenario2, Scenario2_a] := Permission[
    This.Scenario2, Scenario2_a] + Heap[This.Scenario2,
        Scenario2_p2];
//body

```

```

preHeap := Heap;
assert Permission[This.Scenario2, Scenario2.p2] > 0.0;
assert Permission[This.Scenario2, Scenario2.gp2] == 1.0;
Heap[This.Scenario2, Scenario2.gp2] := Heap[This.
    Scenario2, Scenario2.p2];
//post-condition
assert Heap[This.Scenario2, Scenario2.gp2] == preHeap[
    This.Scenario2, Scenario2.p2];
assert Permission[This.Scenario2, Scenario2.gp2] >= 0.5;
Permission[This.Scenario2, Scenario2.gp2] := Permission[
    This.Scenario2, Scenario2.gp2] - 0.5;
assert Permission[This.Scenario2, Scenario2.p2] >= 0.5;
Permission[This.Scenario2, Scenario2.p2] := Permission[
    This.Scenario2, Scenario2.p2] - 0.5;
goto Done_worker2_start;
Done_worker2_start:
//worker2_finish
goto worker2_finish;
worker2_finish:

//pre-condition
if (Permission[This.Scenario2, Scenario2.p2_init] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2.p2_init] := Heap_tmp[This.
        Scenario2, Scenario2.p2_init];
}
Permission[This.Scenario2, Scenario2.p2_init] :=
    Permission[This.Scenario2, Scenario2.p2_init] + 0.5;
if (Permission[This.Scenario2, Scenario2.po2] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2.po2] := Heap_tmp[This.
        Scenario2, Scenario2.po2];
}
Permission[This.Scenario2, Scenario2.po2] := Permission[
    This.Scenario2, Scenario2.po2] + 1.0;
if (Permission[This.Scenario2, Scenario2.gp2] == 0.0)
{
    havoc Heap_tmp;
    Heap[This.Scenario2, Scenario2.gp2] := Heap_tmp[This.
        Scenario2, Scenario2.gp2];
}
Permission[This.Scenario2, Scenario2.gp2] := Permission[
    This.Scenario2, Scenario2.gp2] + 0.5;

```

```

oldHeap := Heap;
havoc Heap;
assume Heap[This.Scenario2, Scenario2.p2_init] == Heap[
    This.Scenario2, Scenario2_gp2];
assume (forall<Ref r, Field f> x :: !(r==This.
    Scenario2 && f == Scenario2.p2_init)) ==> Heap[r, f]
    ] == oldHeap[r, f]);
//body
preHeap := Heap;
assert Permission[This.Scenario2, Scenario2.p2_init] >
    0.0;
assert Permission[This.Scenario2, Scenario2.po2] == 1.0;
Heap[This.Scenario2, Scenario2.po2] := Heap[This.
    Scenario2, Scenario2.p2_init];
//post
assert Heap[This.Scenario2, Scenario2.po2] == preHeap[
    This.Scenario2, Scenario2.p2_init];
assert (Permission[This.Scenario2, Scenario2.a] >= Heap[
    This.Scenario2, Scenario2.po2]) && (Permission[This.
    Scenario2, Scenario2.po2] > 0.0);
Permission[This.Scenario2, Scenario2.a] := Permission[
    This.Scenario2, Scenario2.a] - Heap[This.Scenario2,
    Scenario2.po2];
assert Permission[This.Scenario2, Scenario2.po2] == 1.0;
Permission[This.Scenario2, Scenario2.po2] := Permission[
    This.Scenario2, Scenario2.po2] - 1.0;
assert Permission[This.Scenario2, Scenario2.p2_init] >=
    0.5;
Permission[This.Scenario2, Scenario2.p2_init] :=
    Permission[This.Scenario2, Scenario2.p2_init] - 0.5;
goto Done_worker2_finish;
Done_worker2_finish:
//test C
assert Permission[This.Scenario2, Scenario2.a] == 0.0;
assert Permission[This.Scenario2, Scenario2_gp1] == 0.0;
assert Permission[This.Scenario2, Scenario2_gp2] == 1.0;
}
}

```

Listing A.5.2: Scenario 2 - Code Generator Output

Appendix B

Tables and Figures

HARPO Primitive Type	Boogie Primitive Type
<i>Int8</i>	int
<i>Int16</i>	int
<i>Int32</i>	int
<i>Int64</i>	int
<i>Real16</i>	real
<i>Real32</i>	real
<i>Real64</i>	real
<i>Bool</i>	bool
<i>Perm</i>	perm

Table B.1: Primitive Types Translation Map

<i>accept</i>	<i>acc</i>	<i>as</i>	<i>assert</i>
<i>assume</i>	<i>borrow</i> s	<i>canRead</i>	<i>canWrite</i>
<i>class</i>	<i>claim</i>	<i>co</i>	<i>const</i>
<i>do</i>	<i>div</i>	<i>else</i>	<i>extends</i>
<i>for</i>	<i>gives</i>	<i>ghost</i>	<i>if</i>
<i>implements</i>	<i>in</i>	<i>interface</i>	<i>invariant</i>
<i>mod</i>	<i>new</i>	<i>obj</i>	<i>out</i>
<i>permission</i>	<i>pre</i>	<i>post</i>	<i>private</i>
<i>proc</i>	<i>public</i>	<i>takes</i>	<i>then</i>
<i>thread</i>	<i>type</i>	<i>when</i>	<i>while</i>
<i>with</i>	<i>this</i>	<i>length</i>	<i>forall</i>

Table B.2: HARPO Language Reserved Words

assert	assume	axiom	bool
break	bv0	bv1	bv2
call	complete	const	else
ensures	exists	false	finite
forall	free	function	goto
havoc	if	implementation	int
invariant	modifies	old	procedure
real	requires	return	returns
true	type	unique	var
where	while		

Table B.3: Boogie Reserved Words

Operation	HARPO Operator Symbol	Boogie Operator Type
Addition	+	+
Subtraction	-	-
Multiplication	*	*
Division	/, div	/
Greater Than	>	>
Less Than	<	<
Greater Than or Equal	>_	>=
Less Than or Equal	<_	<=
Not Equal	~=	!=
AND	/\	&&
OR	\/	
NOT	~	!

Table B.4: Operators Types Translation Map

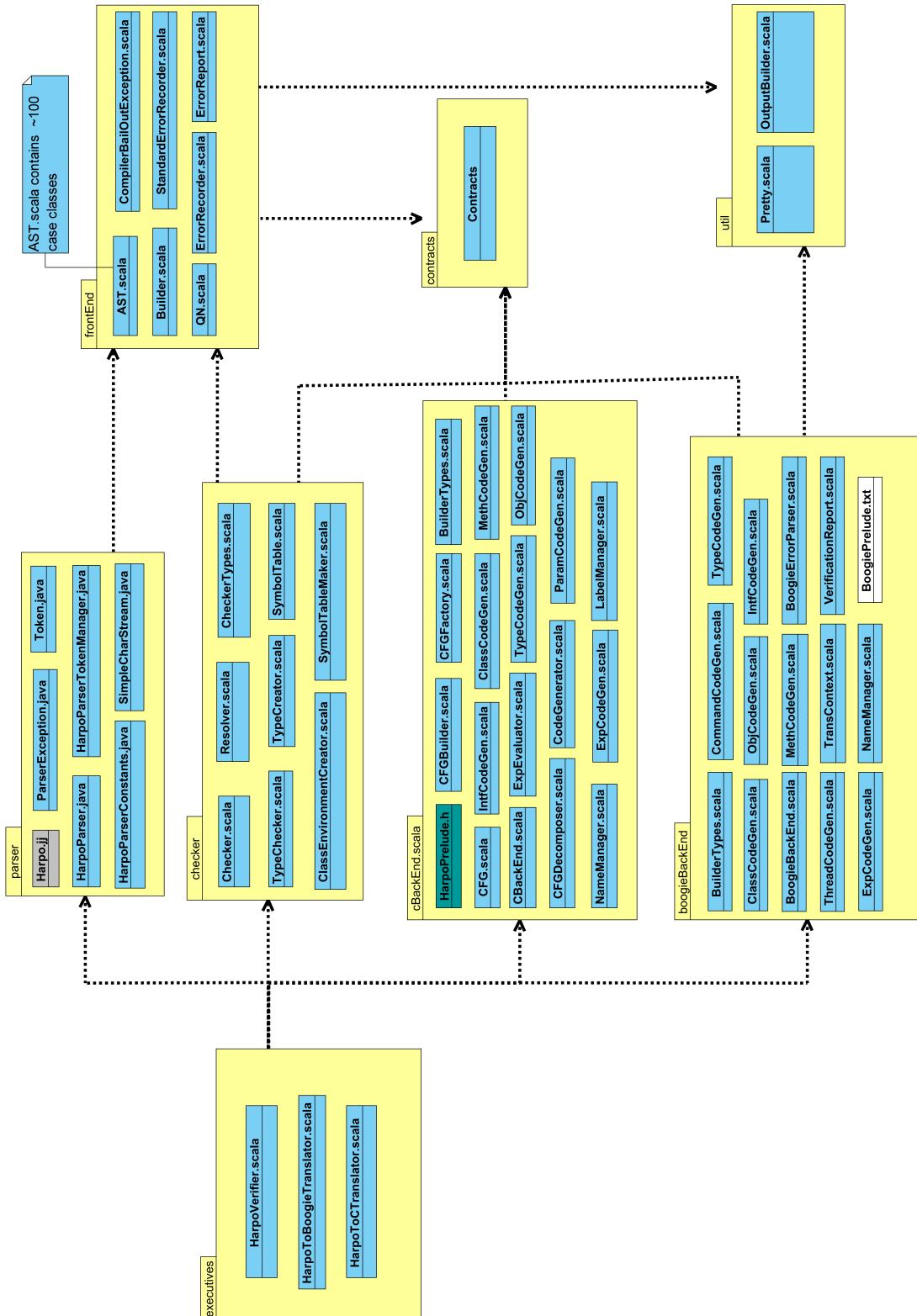


Figure B.1: Complete Package Diagram of HARPO Compiler