Verification of the HARPO Language

by Fatemeh Yousefi Ghalehjoogh

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering (Computer Engineering)

Faculty of Engineering and Applied Science

October 2014

St. John's Newfoundland

Abstract

Verification of sequential programs is hard. Verification of concurrent programs is even harder because it involves considering the possibility of thread interference in addition to the complexity of sequential reasoning.

The purpose of this thesis is to develop a methodology for the automated verification of the multi-threaded and object-oriented HARPO/L language. A verification methodology is presented which allows verifying a program based on its contracts. In particular, it gaurantees data-race-freedom, verification of data invariants (i.e. absence of data corruption), and, to the extent that the pre- and postconditions specify it, correctness of the interface to shared objects. The methodology is built based on implicit dynamic frames with fractional permissions. A specification language is developed based on this methodology to allow programmers to express their design decisions formally. The verification technique is based on verification-condition generation and automated theorem proving. A translation from HARPO/L to the intermediate verification language, Boogie, is provided in the thesis. The efficacy of this approach is demonstrated by translating several examples to Boogie and using automated verification on the translation.

Acknowledgements

I would like to thank my supervisor, Dr. Theodore S. Norvell, for his constant guidance and support throughout my M.Eng. program. This thesis would not have been possible without his insightful advice and patient encouragement.

I would like to dedicate this thesis to my parents.

Contents

List of Figures

List of Listings

Chapter 0

Introduction

0.0 Motivation

The roots of formal verification are in the observation that computer programs can be viewed as mathematical objects with well-determined behaviour [2][3]. In principle, computer programing is considered an exact science in that all the properties of the program and all the results of executing it can be understood from the text of program [1]. As a result, the programs can be modeled as mathematical objects and their correctness can be proved by mathematical methods.

Building reliable software is one of the main goals of engineering. In fact, a program is considered correct if all the design requirements are met by the program [1]. The reality is that programs show errors and it is essential to find errors in the early stages. One of common techniques to achieve this is testing and debugging in which some test cases are performed by user. The fact is, testing takes a considerable time of software projects and they cannot guarantee correctness because tests cover

a limited set of execution cases and testing all cases is infeasible [4]. Given the limitations of testing, more precise approaches are required.

Formal methods have been developed to check the correctness of programs. There have been many advances in formal verifications and several approaches have been developed. One standard approach is generating a collection of logical proof obligations, *verification conditions*, from the program and its specification where the validity of verification conditions implies the correctness of the program. Then, the verification conditions are processed with a theorem prover. Generating verification conditions for high-level programming languages is a complex task. Therefore, this task is split in two steps: First, a transformation of the program into an intermediate verification language (for example, Boogie [5]), and then, a transformation of the intermediate language program into the logical formulas. This methodology creates a level of abstraction over the logical formulas passed to the theorem prover [5].

Reasoning about sequential object-oriented programs is hard, due to aliasing, dynamic binding, modularity and implicit assumptions. Reasoning about concurrent programs is even harder because of data races and possible interference among threads. Therefore, developers must not only consider assumptions for reasoning about sequential programs, they have to consider the effects of other threads. Furthermore, mainstream concurrent programs apply advanced concurrency patterns to increase the performance and flexibility of programs. They use fine-grained locking and concurrent reading to increase parallelism. These patterns make reasoning about concurrent programs more complicated.

0.1 Objectives

In this thesis, we present a verification methodology for the multi-threaded and object-oriented HARPO/L [6] programming language. Our approach is based on implicit dynamic frames [7] with Boyland's fractional permissions [8]. The cornerstone of our methodology is preventing data races. Our approach supports object invariants, rendezvous and fine-grained concurrency. A specification language is developed based on this methodology, which is flexible and allows modular verification.

Based on our methodology, a translation from HARPO/L to an intermediate verification language, Boogie, is provided. Each HARPO/L programming construct is encoded to Boogie and an example is implemented for each case. In the end, some challenging algorithms are specified and verified by this approach.

0.2 Outline

The remaining of this thesis is organized as follows. Chapter 1 provides a background on formal verification and some notable verification methodologies are introduced. In chapter 2, background on HARPO/L and the Boogie intermediate verification language is presented. Chapter 3 describes our verification methodology and specification language. Chapter 4 presents translation of HARPO/L to Boogie language based on our methodology. It includes some examples to show the technical details of translation. In chapter 5, some complex case studies are presented and verified with Boogie. Finally, a conclusion of the thesis with discussion of future work is presented in chapter 6.

Chapter 1

Background on Formal Verification

Formal verification is the process of checking the correctness of programs with formal methods of mathematics. One of the first methodologies was developed by C. A. R. Hoare; it is based on deductive reasoning to generate formal proofs of correctness. One main challenge is the *frame problem*, the problem of formalizing the parts of the heap that remain unchanged by an operation. Since Hoare logic, many studies have been done to solve frame problem. Some of notable proposed methodologies are Separation Logic, Dynamic Frames and Implicit Dynamic Frames which will be discussed in this chapter.

1.0 Hoare Logic

Hoare logic [1] provides a logical basis for reasoning about program correctness. The technique is based on deductive reasoning; it applies sets of valid axioms and valid rules of inference to prove the properties of programs.

Programs are developed based on the design requirements, and it is critical that

programs implement the expected behaviour. To prove this property, the requirements can be specified by making explicit assertions about the values of variables at the end of the program execution and to check whether the implementation conforms to specifications. Furthermore, correctness of a program can depend on the initial values of the variables and these conditions can be specified by assertions too. To formalize this concept, Hoare logic introduced *Hoare triples*. Each Hoare triple states the connection between a program (Q) and its precondition (P) and postcondition (R) with following form:

$$P\ \{Q\}\ R \tag{1.0}$$

The meaning is, if the precondition P is true in initial state, the postcondition, R, will hold after the execution of program Q. This is a partial correctness statement because termination is not considered in Hoare logic.

The assertions are expressed in mathematical notations and the correctness of program is proved by applying axioms and rules. The selection of sets of axioms depends on the properties of the target programming language. Hoare logic was originally developed for a simple imperative language and its axioms and rules cover elementary operations such as addition and multiplication. It also provides inference rules for simple program constructs including assignment, consequence and iteration. For instance, assignment is one of the main features of a programming language. Considering the assignment "$x := e$" the related inference rule is:

$$\frac{}{P[e/x]\ \{x := e\}\ P} \tag{1.1}$$

where x is a variable identifier, P is the postcondition and the precondition $P[e/x]$ is the formula P after substituting e for all free occurrences of x. For example, for the

command "$x := 5$", the following Hoare triple is correct.

$$true\ \{x := 5\}\ x == 5$$

1.0.0 The Frame Problem

Generally, specifications express two kinds of requirements, One is functional requirement describing what type of change is done on the variable and the other is framing requirement describing which variables are allowed to be changed by the statement and other variables are remained unchanged [9]. For instance, considering the previous example, if there is variable y in the program, the postcondition must assert that variable y is not changed by the program:

$$y == 2\ \{x := 5\}\ x == 5 \wedge y == 2$$

The problem with this approach is scalability: for large programs, reasoning about all locations unaffected by a command is a tedious task. This is one of the challenges of writing specification and it is called the *framing problem*.

Moreover, Hoare logic does not consider heap reasoning with aliasing probability. For example, assuming x and y are two variables containing two addresses of a heap, the following statement is not valid because x and y can point to same address. Note that $[x]$ is the dereference of x.

$$[y] == 2\ \{[x] := 5\}\ [x] == 5 \wedge [y] == 2$$

To make it valid, we can add another condition, emphasizing x and y are not equal.

$$[y] == 2 \wedge x \neq y\ \{[x] := 5\}\ [x] == 5 \wedge [y] == 2$$

The reasoning about unaffected locations in heap reasoning has scalability problems too. Many approaches have been developed to deal with the framing problem: they will be explained in the following sections.

1.1 Separation Logic

Separation logic [10] is an extension of Hoare logic to reason about low-level imperative programs that use pointers. It extends the assertions of Hoare logic with predicates describing the heap and introduces new logical operations to express disjointness of heaps and aliasing. The main idea of separation logic is based on *spatial conjuction*, asserting that its sub-formulas hold for disjoint portions of the heap. These extensions allow a concise way to axiomatize pointer operations.

1.1.0 Assertions in Separation Logic

Assertions express the changes on states. States are expressed by two components, a store and a heap. Intuitively, stores describe the content of registers and a heap describes the content of an addressable memory [10]. A store is modeled by a function mapping variables into values, and a heap is modeled with a partial function mapping addresses to values. Two heaps are considered disjoint if they have separate domains. An empty heap is shown by *emp* and a singleton heap with address x and value e is shown as $x \mapsto e$.

The central idea of separation logic is based on a *separating conjuction*, $P * Q$, that asserts that P and Q hold for disjoint parts of heap. A closely related assertion is a *separating implication,* $P -\!\!* Q$. This logic asserts that extending the heap with

a disjoint part that satisfies P results in a new heap that satisfies Q.

1.1.1 The Frame Rule

Separation logic improved Hoare logic by adding new inference rules. To deal with the frame problem, a new rule, with the name of *frame rule* [11] is introduced:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \tag{1.2}$$

where no free variable in R is modified by C. This rule says that it is possible to reason and assert locally about program C and then extend it to specifications with variables and locations which are not modified by C. This property is called *local reasoning* [11]. The idea of local reasoning is that there are many resources in a program, but a statement can work on variables or locations that it has access to and so any initially true conditions involving unaffected locations remain true.

Using the framing rule, extending local specifications is possible. For instance, the previous example can be updated as follows:

$$\{x \to \_ * y \to 2\}\ [x] := 5\ \{x \to 5 * y \to 2\}$$

The possibility of aliasing is removed, and the statement is valid.

1.2 Dynamic Frames

Dynamic frames [9] were developed by Kassios to deal with the frame problem for object-oriented programs. Object-oriented programs introduce new issues such as modularity and encapsulation where previous approaches for verification were designed for non-modular programs.

To illustrate the problem, suppose that program Q is "$x := 1$". We also assume that the program has another variable y. The strongest postcondition must have two assertions: First, the new value of x is one. Second, no other variable changes. As a result, the postcondition will be as the following code:

$$x == 1 \wedge y == \mathbf{old}(y)$$

Such reasoning is not possible in modular programs because, at the time of writing the specification of a module, we do not know all variables, and depending on clients, the program may have different variables.

1.2.0 Modular specifications

Modularity [12] is first introduced by object-oriented programs where a program is divided into independent modules. The advantage of modularity is that any change in a module only affects that module, and there is no need to modify other modules. Consequently, designers of program verification systems have used the modularity concept for program correctness. The idea is that specifications can be written for each module separately, and modules can be verified separately too. Furthermore, modification of a module affects only the proof of that module. Using modular specification, the specification for the previous example is:

$$\mathbf{ensures}\ x == 1\ \mathbf{modifies}\ x$$

The meaning is that the value of variable x is one and the program only modifies variable x, so the specification does not mention about other program variables.

The other property of object-oriented programs is *encapsulation*, which makes

```
class Cell {
    private int x;

    pure int getX()
    { return x; }

    void setX(int value)
    { x := value; }
}
```

Figure 1.0: A class Cell

writing specifications more complicated. Encapsulation means the data representation of the implementation is hidden from clients, and as a result, the specification must be written without mentioning hidden fields. Figure 1.0 shows an encapsulated implementation. The variable x is a private field and clients cannot observe it. Clients can access field x only via setX and getX.

Dynamic frame theory introduced *abstract variables* to be able to reason about encapsulated data. Abstract variables are different from program variables and are only used to reason about hidden fields. In fact, they are used to create a connection between concrete state and the abstract state which is visible to clients. To further simplify this approach, *pure functions* are introduced. A pure function computes a result based on hidden state and has no effect on the state of program. Pure functions are members of a class and are allowed to be invoked in specifications [12].

```
class Cell {
    private int x;

    pure int getX()
        reads footprint();
    { return x; }

    void setX(int value)
        modifies footprint();
        ensures getX() == value;
    { x := value; }

    pure set footprint()
    { return {&x}; }
}
```

Figure 1.1: A class Cell with dynamic frames

1.2.1 Dynamic Frames Implementation

Dynamic frame theory is an approach to specify the *footprint* [12] of statements in an abstract way. A footprint is a set of locations which is accessed by a method or function. Footprints are encoded by pure functions which are called *dynamic frames*. Moreover, methods and functions are specified by those pure functions. Now, the previous example can be annotated using pure functions as shown in figure 1.1.

The pure function footprint is the footprint of class C. The method setX is annotated by modifies clause; means the method setX is allowed to access the fields returning by footprint. Furthermore, the function getX is annotated by reads clause; meaning it only depends on fields returning by footprint.

This approach solves the frame problem by allowing specifications to explicitly

declare the part of the heap they may modify, by functions of the heap. It guarantees that when a variable is changed, the other frames that are disjoint from the frame of the changed variable remained unchanged. The computed frames are considered dynamic because the returning values of these functions may change as the heap is modified [13].

1.3 Implicit Dynamic Frames

The Dynamic frames methodology is a powerful and flexible way to specify and verify object-oriented programs. However, it requires that frame annotations be written for each method, and be checked at verification time. Implicit dynamic frames tackles this problem by introducing a new specification style with *access permissions*, which eliminates the need to explicitly write and check the frame annotations. Instead, the frame information can be inferred implicitly from specifications. The new annotations are more concise and discharge fewer proof obligations [7].

The role of access assertions is to represent permission to access a location. A method can access a memory location if it has permission to do so. The permissions required by the precondition implicitly define the upper bound of locations modified by the method.

Accessibility to location x is denoted by $\mathsf{acc}(x)$. Suppose a method needs to read or write a memory location; in order to be allowed to do so, the location must be accessible by the method's precondition. Consequently, the location will be accessible during execution of the method. Figure 1.2 shows an example of implicit dynamic frames.

```
class Cell {
    Cell()
        ensures acc(x) /\ x = 0;
    { this.x := 0; }

    void setX(int v)
        requires acc(x);
        ensures acc(x) /\ x = v;
    { x := v;}
}
```

Figure 1.2: A class Cell with implicit dynamic frames

The method setX modifies x, so its precondition requires accessibility of location x. In other words, acc(x) in precondition transfers permission to access x from the caller to the callee and acc(x) in the postcondition returns permission to the caller. Similarly, the constructor changes the fields of the new object and does not require access permission because when a new object is created, the permission is transferred from the system to the constructor.

This approach solves the frame problem for modular programs. The method can be improved to support data abstraction. *Pure methods* are introduced to have data abstraction; pure methods are side-effect free methods that can be used in contracts and expressions. There are two kinds of pure methods: normal pure methods and predicates. A normal pure method abstracts over an expression, while a predicate abstracts over an assertion. The previous example can be improved by hiding permission on field x as shown in figure 1.3.

The predicate valid and the pure method getX are added. Instead of directly using

```
class Cell {
    Cell()
        ensures valid() /\ getX() = 0;
    { this.x := 0; }

    void setX(int v)
        requires valid();
        ensures valid() /\ getX() = v;
    { x := v;}

    predicate bool valid()
    { return acc(x); }

    pure int getX()
        requires valid();
    { return x; }
}
```

Figure 1.3: A class Cell with data abstaction

the access permission, the predicate valid is called in method contracts. Predicates are self-framing and valid does not have preconditions. The pure method getX has a precondition that requires access on location x. Since pure methods never modify the state, the return value of getX only depends on locations required to be accessible by its precondition.

1.4 Verification of Concurrent Programs

Multiprocessor computers allow multiple application programs to execute simultaneously or a single program to execute on multiple processors using multithreading [14]. Multithreading improves the performance and flexibility of programs, but also

complicates reasoning. Reasoning about sequential object-oriented programs is hard, due to aliasing, implicit assumptions and modularity. Reasoning about concurrent programs is even harder due to their having more complicated behaviour.

A concurrent program consists of multiple processes (computations) executing together to perform a task. Threads are processes in a shared resources system. The characteristics of multithreaded programs make it hard to write correct programs. First of all, the scheduling of threads is non-deterministic, which makes it difficult to test or reproduce the behaviour to find errors. Secondly, reasoning about multithreaded programs is complicated since the interference of other threads has to be considered. In particular, threads can access a shared resource simultaneously [15].

1.4.0 Properties of Concurrent Programs

To verify the correctness of concurrent programs, two essentially kinds of properties of the program must be proved: *safety* properties and *liveness* properties. A safety property states that something bad can not happen. For example, partial correctness is a safety property. It means that the program can not stop in an incorrect state. A liveness property states that something good will eventually happen. Program termination is an example of liveness property. Verifying concurrent programs requires techniques to prove safety and liveness properties [16].

One important category of safety properties is *mutual exclusion* [14]. It states that no two processes are in their critical sections at the same time. Mutual exclusion can be used to avoid *data races*. The problem of data races will be discussed in the next part and then it will be explained how to implement mutual exclusion to avoid data

races.

Data Race

A data race happens when two threads access a shared memory location concurrently and one of the accesses is a write access. This is problematic if shared data has to be accessed in a consistent way. For instance, assuming reading and writing are not atomic, if one thread reads a memory location while the other one is trying to write, the thread might read a corrupt value. Also, if two threads try to change a memory location simultaneously, the resulting value can be corrupted. An example data race is the following program:

$$x := x + 1 \;||\; x := x + 2$$

Two threads compete to access a shared variable. The meaning of statements is dependent on the ordering of interleaving and also the granularity of operations [17]. If each of the assignment statements is an atomic action then there are two interleaving of actions and two results are possible. If the assignment command consists of two or more atomic actions, then more interleavings and results are possible [18].

Data race is a programming error. It is very difficult to detect data races by testing techniques. In fact, race freedom is a basic building block for writing and verifying multithreaded programs. Race free programs guarantee desired behaviours of programs and free us from considering the details of interleaving and granularity [17]. Data races can be avoided by properly synchronizing process interactions. One way to avoid data race is to have *mutual exclusion groups* in critical sections.

Mutual Exclusion

In a shared memory system, synchronization allows controlled communication be-

tween threads. Mutual exclusion groups are a common way to synchronize concurrent threads. A mutual exclusion group is a set of commands such that not more than one command in the group can be executing at once [17]. This approach greatly simplifies writing and reasoning about concurrent programs by eliminating the need to consider the details of interleavings and granularity. A standard way to achieve mutual exclusion is to use locks. A lock ensures that one or more sections of code is not executed concurrently by multiple threads by requiring threads to first acquire the lock and if it was held by another thread, the requesting thread will wait until it becomes available. Locks can be *fine-grained* or *coarse-grained* [14].

Traditional concurrent implementations use coarse-grained locking in which a single lock is used to guard an entire data structure, such as a linked list or a tree [19]. This approach eliminates inconsistent executions and reasoning is fairly easy. However, it limits concurrency. For instance, modifying an element of a tree requires that the entire data structure be locked. This reduces parallelism and it is more efficient to lock only the elements which are required.

Fine-grained locking allows more concurrency by allowing multiple threads to work on same data structure simultaneously. This technique associates each object with its own lock and threads working in different parts of a data structure do not need to exclude each other. For instance, each node of a tree could be protected by its own lock. In this approach, atomicity is achieved by acquiring a set of locks instead of a single lock [19]. This approach can have higher efficiency but complicates reasoning about concurrent programs.

1.4.1 Verification Methodology for Concurrent Programs

The verification methodology for concurrent programs has to support the complications of accessing the memory by multiple threads. In particular, it has to rule out data races and support multiple readers and fine-grained locking [19]. Implicit dynamic frame theory can handle the data race issue. However, it only supports full access permission, meaning a thread either can read and write a memory location or it does not have any permission at all. As a result, if a thread only reads a memory location, it must have full access. Instead, we can think of two types of permission: *write permission* and *read permission*. Write permission must be unique but read permission can have multiple copies excluding any write permission. One way to have multiple concurrent readers and fine-grained locking is *fractional permissions* [8]. Extending implicit dynamic frames with fractional permissions enables verifying concurrent programs. *Chalice* [19], a language and verifier for concurrent programs implements a similar approach.

1.4.2 Fractional Permission

Fractional permission [8] is a popular approach in reasoning about concurrent programs. It can prevent data races while allowing multiple readers. The model associates one (whole) permission to each memory location and the whole permission is needed to modify the memory location. Permission can be held or transferred between threads and objects. Also, permissions can split into fractions. A non-zero fraction is required to read a memory location. This allows multiple readers. Finally, fractions can be joined back together, and when all readers have returned their fractions, it

```
class Cell {
    var a, sum : int;

    method Add()
        requires acc(sum, 1) /\ acc(a, 0.5)
        ensures acc(sum, 1) /\ acc(a, 0.5)
    { sum := sum + a; }
}
```

Figure 1.4: A class Cell with fractional permissions

is possible to write the memory location. Fractional permission can only be transferred, split or combined, but never duplicated. As a result, at most one thread can have write permission at a time and sum of permissions of threads for each location is one whole permission. This property ensures race freedom and allows arbitrary concurrent reads [20].

To verify modular concurrent programs, fractional permissions can be combined with implicit dynamic frames. In this solution, permissions are abstractly a quantity between 0 and 1. A permission of 1 means the thread has write permission and any non-zero permission means the thread has read permission. Each method specifies its required permission in its precondition and the returning permission in its postcondition. Figure 1.4 shows an example of fractional permissions.

The method Add requires full permission for the location sum and read permission for the location a. The execution thread must have full permission for location sum and at least 0.5 permission for location a. Upon a call to Add, the required permissions are transferred from the caller to the callee and after the call, the permissions specified in the postcondition transfer from the callee to the caller.

Chapter 2

Background on HARPO/L and Boogie

In this chapter, technical background about the HARPO/L programming language and the Boogie language, which will be needed in the following chapters, is presented.

2.0 HARPO/L

2.0.0 Introduction

Digital hardware has improved rapidly regarding speed, density and efficiency making it possible to implement applications on Integrated Circuits (ICs). ICs can be divided in three groups, non-programmable ICs such as traditional ASICs, programmable ICs such as microprocessors, and reconfigurable hardware such as FPGAs and CGRAs. Abstract designing methods, free of implementation technology, allow to deal with a wide range of implementations. The HARPO/L language is designed to achieve this

objective [21].

Application-Specific Integrated Circuits (ASICs) are designed for a particular use, and operate on a high performance level. However, they have low flexibility and any modification leads to redesign of the whole circuit. In contrast to ASICs are microprocessors. Microprocessors are general purpose systems with fixed hardware, which can be programmed for many applications. They have lower efficiency than nonprogrammable ICs but higher flexibility. Reconfigurable architectures are the modern technologies which fill the gap between nonprogrammable hardware and microprocessors. Field-Programmable Gate Arrays (FPGAs) are fine-grain reconfigurable architectures and microprocessors. Their architecture consists of reconfigurable blocks and reconfigurable interconnections at bit level which can be configured by the user. FPGAs have the advantage of high flexibility, but due to their fine-grained structure they can have low performance. On the other hand, Coarse-Grained Reconfigurable Architectures (CGRAs) have higher performance. They have word length processing units which overcome the performance issues of FPGAs [22].

HARPO/L is a behavioral language to create hardware configurations for a wide range of platforms including configurable hardware and microprocessors. The language is object oriented, supporting parallel and concurrent computing. However, to support hardware characteristics, it has features which make it different from typical high-level programing languages.

The first difference is the representation of objects. In hardware implementation, an object is represented by a concrete block of gates which is configured at construction time. As a result, all objects in HARPO/L are formed statically at configuration time and there is no dynamic memory allocation. Also, the connections between

objects are generated at configuration time and, consequently, there is no reference assignment in HARPO/L. The second difference is the implementation of methods; in hardware, a method call is a set of gates and connections which implements the body of a method. As a result, HARPO/L does not allow multiple threads to work on the same method concurrently. Finally, hardware is inherently parallel and allows higher level of parallelism than microprocessors. To support this feature, HARPO/L has a **co** statement which allows explicit parallelism [23].

The features of HARPO/L language are introduced in this section. This section is based on [6].

2.0.1 Primitive types

Primitive types are build-in types. In HARPO/L, the primitive types are scalar types including Integer, Real and Boolean. Integral and real types are represented in different ranges, as follows

Integer: int8, int16, int32, and int64

Real: real16, real32, real64

Objects of primitive types can be assigned several times. Shorter types are sub-types of longer ones. Therefore, the shorter types can be assigned to variables with longer types and the type of an expression is determined by the longer type.

2.0.2 Arrays

An array is a collection of locations or objects with same type. In HARPO/L, arrays are single-dimensional and have fixed sizes. The language allows to have an array of

arrays too. An array is defined as follows:

$$\begin{aligned} \mathit{ArrayDecl} &\rightarrow \textbf{obj}\ \mathit{Name}\ \underline{[}:\mathit{Type}\underline{]} := \mathit{ArrayInit} \\ \mathit{ArrayInit} &\rightarrow (\textbf{for}\ \mathit{Name} : \mathit{Bounds}\ \textbf{do}\ \mathit{InitExp}\ \underline{[}\textbf{for}\underline{]}) \end{aligned} \tag{2.0}$$

Each item has a position called its index and can be accessed by the name of object and the index of the item.

2.0.3 Classes and Fields

Classes are the basic construct in HARPO/L. Classes can be instantiated several times to create objects, but objects cannot be assigned after initialization. Class members include fields, methods, and threads. Each class has one constructor. The constructor parameters can be either constants with keyword **in**, or objects to which this object is connected with keyword **obj**.

$$\begin{aligned} \mathit{ClassDecl} \rightarrow\ & (\textbf{class}\ \mathit{Name}\ \mathit{GParams}^{?}\ (\mathit{CParam}^{+,})\ \underline{(}\textbf{implements}\ \mathit{Type}^{+,}\underline{)}^{?} \\ & \underline{(}\mathit{ClassMember}\underline{)}^{*}\ \underline{[}\textbf{class}\ \underline{[}\mathit{Name}\underline{]}\underline{]}) \end{aligned} \tag{2.1}$$

In the following code, class Adder is declared. Object a with type of the class is created. Object a is connected with two other objects i and j at compile time. Moreover, there will be no reassignment to the object a after construction.

```
(class Adder
    constructor(obj i : Int16, obj j : Int16)
    public proc Add()
        ......
class)
obj a := new Adder(i, j);
```

Fields are objects defined in a class or interface. A field must have an access modifier. A private field is accessible inside its class but a public field can be accessed by other classes. Fields are declared in the following form:

$$\begin{aligned} Field &\longrightarrow Access^{?}\ \textbf{obj}\ Name\ \underline{[}:Type\underline{]} := InitExp \\ Access &\longrightarrow \textbf{private} \mid \textbf{public} \end{aligned} \tag{2.2}$$

The *Type* can be omitted; in this case, the type of the object is determined by the type of *InitExp*. If the *Type* is declared by the user, *InitExp* must have same type as the object.

2.0.4 Interfaces

In the HARPO/L language, an interface is a type. The members of an interface can be fields or method declarations. The implementation of a method declaration must be inside a thread of a class that implements the interface.

$$\begin{aligned} IntfDecl &\rightarrow (\textbf{interface}\ Name\ GParams^{?}\ \underline{(}\textbf{extends}\ Type^{+,}\underline{)}^{?} \\ &\qquad \underline{(}IntfMember\underline{)}^{*}\ \underline{[}\textbf{interface}\ \underline{[}Name\underline{]}\underline{]}) \\ IntfMember &\rightarrow Field \mid Method \mid ; \end{aligned} \tag{2.3}$$

In the following example, an interface is declared with two public methods.

```
(interface Buffer
    public proc put(in value : Int32)
    public proc get(out value : Int32)
interface)
```

2.0.5 Threads and Blocks

Threads are executable code blocks of a class. Each class has zero or more threads which are executed as a result of object creation.

$$Thread \rightarrow (\textbf{thread}\ Block\ [\underline{\textbf{thread}}]) \tag{2.4}$$

Threads enable concurrency in an object, with multiple threads processing at the same time. A block is a sequence of statements:

$$Block \rightarrow CommandBlock \mid LocalDeclaration \mid ; Block \mid \tag{2.5}$$

2.0.6 Statements

Statements are declared within threads. The grammar of HARPO/L statements are described in this part.

- **Local Variable Declaration**

A local variable represents an object with a value. The scope of a local variable is the block it is declared.

$$ObjectDecl \rightarrow \textbf{obj}\ Name\ [\underline{:\ Type}] := InitExp \tag{2.6}$$

- **Assignment**

In HARPO/L, objects with primitive types are allowed to be assigned. Aditionally, multiple objects can be assigned at one time. The number of objects must be the

same as the expressions in the right hand side.

$$\begin{aligned} Command &\rightarrow ObjectIds := Expressions \\ ObjectIds &\rightarrow \underline{(}ObjectId\underline{)}^{+,} \\ Expressions &\rightarrow \underline{(}Expression\underline{)}^{+,} \\ ObjectId &\rightarrow Name \mid ObjectId[Expression] \mid ObjectId.Name \end{aligned} \tag{2.7}$$

- **Sequential Control Flows**

Sequential control flows are similar to other high-level languages. They include **if** statement for alternation, **while** and **for** statement for repetition structure.

$$\begin{aligned} Command \rightarrow\ & (\textbf{if}\ Expression\ \underline{[}\textbf{then}\underline{]}\ Block\ \underline{(}\textbf{elseif}\ Expression\ \underline{[}\textbf{then}\underline{]}\ Block\underline{)}^{*} \\ & \quad \underline{(}\textbf{else}\ Block\underline{)}^{?}\ \underline{[}\textbf{if}\underline{]}) \\ & \mid (\textbf{while}\ Expression\ \underline{[}\textbf{do}\underline{]}\ Block\ \underline{[}\textbf{while}\underline{]}) \\ & \mid (\textbf{for}\ Name\ :\ Bounds\ \underline{[}\textbf{do}\underline{]}\ Block\ \underline{[}\textbf{for}\underline{]}) \end{aligned} \tag{2.8}$$

- **Method Implementation**

Methods are used for thread synchronization. A method has a declaration part in the class and an implementation block located in the thread. A method may have guard. A guard is a Boolean expression which must be true to execute the method.

$$\begin{aligned} command &\rightarrow \Big(\textbf{accept}\ MethodImp\ \underline{(}\mid MethodImp\underline{)}^{*}\ [\textbf{accept}]\Big) \\ MethodImp &\rightarrow Name(\underline{(}Direction\ Name : Type\underline{)}^{*,})\ \underline{[}Guard\underline{]}\ Block_0\ \underline{[}\textbf{then}\ Block_1\underline{]} \\ Direction &\rightarrow \textbf{in} \mid \textbf{out} \\ Guard &\rightarrow \textbf{when}\ Expression \end{aligned} \tag{2.9}$$

Methods in HARPO/L have different behavior from other high level programming languages. Because of the hardware specifications, HARPO/L methods implement rendezvous. A method is implemented inside a thread and, when the thread reaches the accept statement, it waits for a call for one of its implemented methods. Once there is at least one call and the guard is true, the corresponding method body may be executed.

In this methodology, one call at a time is served and other calls must wait until the sequential execution flow of the server thread reaches the accept statement. Furthermore, a thread cannot call a method implemented by itself, as this results a deadlock. Rendezvous can be used to model other synchronization mechanism such as semaphores.

- **Lock**

Locking is a fine grained technique to have data consistency. In concurrent programs, shared locations may be accessed by several threads at the same time. Accesses are either read accesses or write accesses. Data access is not necessarily atomic and takes a span of time. To avoid data inconsistency, while writing to a location, other read or write accesses must be prevented. Locks are one synchronization mechanism to have mutual execlusion.

$$Command \rightarrow \Big(\textbf{with}\ Exp\ \underline{[}Guard\underline{]}\ \underline{[}\textbf{do}\underline{]}\ Block\ \underline{[}\textbf{with}\underline{]}\Big) \tag{2.10}$$

In this statement, *Exp* must be an object implementing interface Lock. The guard is a Boolean expression.

- **Parallelism**

The **co** statement gives the feature of explicit parallelism, with it the programmer can define parts of program to be executed concurrently.

$$\begin{aligned} Command \;\rightarrow\; & \Big(\mathbf{co}\ Block\ \underline{(||Block)}^{+}\ \underline{[\mathbf{co}]}\Big) \\ & \mid \Big(\mathbf{co}\ Name : Bounds\ \underline{[\mathbf{do}]}\ Block\ \underline{[\mathbf{co}]}\Big) \end{aligned} \tag{2.11}$$

In parallel accesses, it is the responsibility of the programmer to ensure data consistency by using synchronization methods.

2.0.7 Expressions

The grammar of expressions in HARPO/L is as follows:

$$\begin{aligned} Exp &\rightarrow Exp0\ \underline{(}\underline{(} => \mid <= \mid <=> \underline{)}\ Exp0\underline{)}^{*} \\ Exp0 &\rightarrow Exp1\ \underline{(}\underline{(}\backslash/ \mid or\underline{)}\ Exp1\underline{)}^{*} \\ Exp1 &\rightarrow Exp2\ \underline{(}\underline{(}/\backslash \mid and\underline{)}\ Exp2\underline{)}^{*} \\ Exp2 &\rightarrow Exp3 \mid not\ Exp2 \mid {\sim}Exp2 \\ Exp3 &\rightarrow Exp4\ \underline{(}\underline{(} = \mid {\sim} = \mid < \mid \_ < \mid > \_ \mid >)\ Exp4\underline{)}^{*} \\ Exp4 &\rightarrow Exp5\ \underline{(}\underline{(} + \mid - \underline{)}\ Exp5\underline{)}^{*} \\ Exp5 &\rightarrow Primary\ ((* \mid / \mid \mathbf{div} \mid \mathbf{mod})\ Primary\underline{)}^{*} \\ Primary &\longrightarrow (Exp) \mid - Primary \mid ObjectId \end{aligned} \tag{2.12}$$

2.1 Boogie

A standard approach to program verification is based on theorem proving techniques. In this approach, the input program and its specification are transformed to logical formulas, called *verification conditions*. The validity of verification conditions shows the correctness of program. The verification conditions are then passed to a theorem prover and discharged automatically.

Generating verification conditions is a complex task involving a great number of decisions. As a result, this task is accomplished in two steps. First, the input program and its specifications are translated to an intermediate programming language (Boogie), and then the intermediate-language program is transformed to verification conditions.

In this section, we give an introduction to Boogie, an intermediate language for program verification. Boogie is an imperative language, consisting of two types of declarations: mathematical constructs and imperative constructs. The mathematical part features types, constants, functions, and axioms. The imperative parts consist of global variables and procedure declarations. This section outlines the features of Boogie language. This part is based on [5].

2.1.0 Types

A type is either a built in type or user-defined type. Built-in types consist of primitive types including **int** and **bool** and (possibly polymorphic) map type also known as array types. For instance, each individual of the type "[*ref*] **bool**" maps *ref* individuals to Booleans.

Boogie allows type constructors. The following example defines a type constructor with the name of *Multiset* and it has one argument.

type $Multiset\ \alpha$;

Two instantiations of the *Multiset* type are *Multiset* **int** and *Multiset* **bool**.

2.1.1 Functions

Function declarations specify mathematical functions. For instance, the following code declares a function intended to return the length of an input string.

function $length$(**string**) **returns**(**int**);

2.1.2 Axioms

Properties of constants and functions are declared by axioms. For example, the following code declares that the function *length* returns 10 for every s.

axiom $length(s) == 10$;

2.1.3 Global variables

Global variables represent components of all states and can be changed by all procedures. They must have distinct names from constants and other global variables. For example, the following code declares a variable to hold current *sum*.

var sum : **int**;

2.1.4 Execution traces

An execution trace is a nonempty sequence of states. A finite and terminating execution trace is called well-behaved program execution. An execution trace which goes wrong after a finite number of states is called ill-behaved program execution. If an execution trace continues infinitely, it is called diverged and is a well-behaved nonterminating execution trace.

2.1.5 Procedures and Implementations

A procedure declaration defines sets of execution traces. There are two kinds of execution traces, *caller traces* and *callee traces*. These traces are determined by the signature and specification of a procedure. For example,

procedure $Add()$;
    requires $x > 0$;
    ensures $sum > 0$;
    modifies x;

declares a procedure with specifications. A procedure can have three types of specification, *precondition*, *postcondition* and *modification*. A precondition is declared by **requires** clause which is a Boolean condition. The caller is responsible to establish the precondition and the callee can assume it to be true in initial state of execution trace of procedure execution. A postcondition is declared by an **ensures** clause which, is a Boolean condition. The procedure implementation is responsible for establishing it in final state of execution trace of procedure, and the caller can assume the postcondition is true at the time of return. The **modifies** clause specifies the potentially modified global variables in the body of procedure implementation. Mul-

tiple **requires** or **ensures** clauses are equal to one **requires** or **ensures** clause. Also, multiple **modifies** clauses are equivalent to one **modifies** clause.

A procedure implementation is a body of code representing a set of execution traces. For example,

implementation *Add* (x : **int**)
{
    $sum := sum + x$;
}

A procedure implementation is correct if its set of execution traces is a subset of the execution traces of procedure specification. The postcondition and implementation body are allowed to use **old**(E) expression which represents the value of expression E on entry to the procedure

2.1.6 Statements

Statements are used in procedure implementation. Local variables are declared at the beginning of procedure implementations followed by statements. Boogie has statements similar to other imperative languages as well as statements for verification and specification purposes, such as loop invariant syntax, **assert**, **assume** and **havoc** statements. The grammar of Boogie statements is described in this part.

- **Assignment Statement**

An assignment modifies the value of a list of variables in parallel. First, right-hand expressions are evaluated and then the mutable variables in left-hand are assigned in parallel. For example, the following code swaps the value of x and y.

$x, y := y, x$;

The statement $a[i] := E$; is a map update. It changes map variable a so that the new value maps i to E.

- **Assertions and Assumptions**

Assertion and assumption are used to encode proof obligations. An assertion expresses a verification condition which must hold in every well-behaved execution trace. If the assertion does not hold, the execution goes wrong and it is considered an error. It can be used to check conditions. For example, in translating of a division assignment, $x = y \ / \ z$; it may be checked that z is not equal to zero as follows:

assert $z \neq 0$; $x = y \ / \ z$;

Assumptions hold in every feasible execution trace. In fact, an **assume** statement restricts feasible traces. If the assumption holds, it is considered a no-op. If the assumption does not hold, that trace is considered infeasible and the already taken steps are removed. Some important uses of assumption are in **if** statement and the **havoc** statement. The **assume** statement helps the verifier to render infeasible traces.

- **Havoc**

The **havoc** statement takes a list of variables and assigns arbitrary values to them. These values are chosen blindly but they are restricted by the types of variables, where conditions and program axioms. For instance, the following example sets x to a non-negative value. The **axiom** declares some properties about the program's constants and functions.

axiom $x >= 0$;
....
havoc x;

The **havoc** statement operates on entire variable. For instance, to assign arbitrary value to parts of a mapping variable, a temporary variable may be used.

var tmp : **int**;
. . . .
havoc tmp;
$a[i] := tmp$;

It is frequent that **havoc** is followed by an **assume** statement. The **assume** statement introduces an assumption in the program to be verified. The effect of **assume** statement is limiting the value chosen by the **havoc** statement. For example, the following code sets x so as to satisfy the condition $0 <= x$.

havoc x; **assume** $0 <= x$;

- **Label Statements and Jumps**

A label statement indicates a program point. It can be used with **goto** and **break** statements. A **goto** statement transfers the control of the program to the specified label. A **goto** statement can have more than one label where the choice between them is done randomly. The specified labels by **goto** must reside in the same implementation body. The following example shows a **goto** statement.

$i := 0$;
while $(i < 20)\{$
    if $(a[i] == N)$ $\{$**goto** $Done;\}$
    $i := i + 1$;
$\}$
$Done$:

A **break** statement transfers control of program to the statement following the enclosing statement. Any **break** statement with no label, transfers the control of program to the nearest enclosing **while** loop. Any break statement with specified

label, transfers the control of program to the immediate statement that follows the label. The following example shows a **break** statement.

$i := 0;$
while $(i < 20)\{$
    if $(a[i] == N)$ $\{$**break**;$\}$
    $i := i + 1;$
$\}$

- **If Statement**

The **if** statement is similar to the usual conditional statement. There are two kinds of **if** statement in Boogie. The first kind uses a Boolean expression to choose one of its alternatives.

if (E) Thn **else** Els;

The second kind uses a * as the *WildCardExpr* and chooses arbitrarily between the alternatives.

if $(*)$ Thn **else** Els;

- **While Statement**

The **while** loop is the usual iteration that loop body executes while the guard is true. The syntax is as follows:

while (E) $invs$ S;

Also, Boogie supports another form of **while** loop which iterates for an arbitrary number of times.

while $(*)$ $invs$ S;

Both while loops have an invariant declaration. The loop invariant must hold at the start of each iteration. The loop invariant is checked immediately before the loop and at the end of the loop body.

- **Call Statements**

A call statement represents the caller traces defined by the procedure declaration. The following example calls procedure P.

call $ys := P(xs)$;

The xs are the input parameters and ys are the output parameters. The ys list must be distinct and mutable variables. The number of parameters is the left-hand-side must be equal to the number of formal out-parameters of the procedure. The procedure declaration can have generic types, in this case, the type arguments will be instantiated in call statement too.

Chapter 3

Verification Methodology

HARPO/L is a multi-threaded object-oriented programming language. It makes use of mutable objects, aliasing and modularity. It also supports multithreading, which imposes the need to consider data races. We are interested in verifying HARPO/L programs with such characteristics. This requires developing a system of specifications and a methodology of reasoning.

Specifications are sets of annotations describing the behavior of a program. More specifically, they explicitly declare the requirements of implementation. Writing specifications can be done in many ways but we are interested in developing a systematic way to write specifications and to reason about specifications and programs. Our verification methodology is based on implicit dynamic frames [7] with fractional permissions [8]. This model allows modularity, fine-grained locking, concurrent data structures, rendezvous and concurrent readers. More importantly, it allows ruling out data races. In this section, the specification language and verification methodology will be described.

3.0 Pre- and Postconditions

Each method is specified by *pre-* and *postconditions*. The precondition declares the constraints on the state which must be valid before execution of the method. A postcondition declares conditions which are guaranteed to be correct after method execution. The precondition must be established by the caller and the postcondition must be established by the method. Both preconditions and postconditions are Boolean and side-effect-free expressions. A method can have more than one precondition or postcondition. The final precondition is the conjunction of the preconditions and the final postcondition is the conjunction of the postconditions. The declaration of precondition and postcondition contracts is as follows:

$$\begin{aligned} &\mathit{MethodDecl} \longrightarrow \mathit{Access}\ \textbf{proc}\ \mathit{Name}\ (\mathit{Parameters})\ \underline{(}\mathit{ConditionSpec} \mid \mathit{PermissionSpec}\underline{)}^{*} \\ &\mathit{ConditionSpec} \longrightarrow \textbf{pre}\ \mathit{precondition} \\ &\qquad\qquad\qquad \mid \textbf{post}\ \mathit{postcondition} \end{aligned}$$

To illustrate how method contracts work, the example in listing 3.0 is provided. Class Math has a method called divide which calculates the division of two input numbers a and b and returns the result c. The method is annotated with two specifications. The precondition, starting with the keyword **pre** indicates that the method divide expects the caller to provide a non-zero value for b. The postconditon, starting with the keyword **post** declares the output parameter c will equal to division of input parameters.

The postcondition is a two-state predicate meaning it relates the value of variable at entry to method to the final value of the variable at the exist. The final value is marked with an apostrophe to distinguish it from the initial value.

(**class** Math

```
    public proc divide(in a, b : real32, out c : real32)
    pre b != 0
    post c' = a / b

    (thread (*t0*)
        (while true
            (accept divide(in a, b : real32, out c : real32)
               c := a / b
            accept)
        while)
    thread)
class)
```

Listing 3.0: A program that illustrates pre- and postcondition

3.1 Ghost States

Ghost variables are added to aid reasoning about the code. They are marked by the **ghost** keyword which distinguishes them from program variables. They have no effect on program execution and we can remove them without affecting the program behavior. Ghost variables may be used in specifications and ghost code. More specifically, they may be used as fields, local variables, constructor parameters or method parameters. Ghost variables can be updated by ghost code. The ghost variables allows writing specifications for modular verification. The declaration of ghost variables is shown below:

$ObjectDecl \longrightarrow [\textbf{ghost}]\ (\textbf{const} \mid \textbf{obj})\ Name\ [\ :Type]\ :=\ InitExp$
$CParam \longrightarrow [\textbf{ghost}]\ \textbf{obj}\ Name : Type \mid [\textbf{ghost}]\ \textbf{in}\ Name : Type$

3.2 Permissions

Threads can access memory locations concurrently. In order to avoid data races and have multiple concurrent readers, data accesses must be restricted, meaning, threads may access memory locations only if they are allowed. This model is implemented by fractional permissions [8]. In this methodology, a thread may read or write a memory location only if it has *read* or *write* permission for that location. A permission is a quantity between 0 and 1. To write a memory location, a thread requires having full permission which is one. To read a memory location, a thread needs a permission greater than zero. The sum of permissions for each location in whole system is less than or equal to one. This eliminates data races but allows multiple readers.

For verification purpose, we add a new primitive type **Perm** which represents real values between zero and one. Zero represents no permission, one represent full permission and any value between zero, and one means read permission. This type is added for verification purposes, and it can only be used for ghost variables.

Pemission specifications declare the amount of permission on locations. The syntax of permission maps in specifications is shown below:

$PermMap \longrightarrow LocSet \mid LocSet@Expression \mid PermMap, PermMap$
$LocSet \longrightarrow ObjectId \mid \{Name : Set\ [Guard]\ \textbf{do}\ LocSet\}$
$Set \longrightarrow \{Expression, ..Expression\} \mid \{Expression, .., Expression\} \mid Expression$
$ObjectId \longrightarrow Name \mid ObjectId[Expression] \mid ObjectId.Name$

To express the amount of permission on locations the symbole @ is used. For instance, $o.f@0.5$ declares the amount of permission on location $o.f$ is $o.5$. If the amount of permission is full permission, it is allowed to remove the amount part. A location set may be a single object id or a set of locations. The latter form is useful for

expressing permission on array locations. For instance, $\{i : \{0, .., 10\}$ **do** $E.f\,[i]\}$ @0.5 declares 0.5 permission on array elements numbered 0 to 10, inclusive. As shown in the productions, set has three forms. In the first form, the higher bound is exclusive. In the second form, the higher bound is inclusive. The third form is equivalent to $\{0, ..Expression\}$.

3.3 Threads

A thread can access a memory location if it has permission. Initially, threads can get access to a memory location by **claim** specification. This gives the specified permission to the thread which can be read or write permission. We can assume that these permissions are transferred from the system to the threads at the beginning of execution. The sum of claims for each location over all permission holders must be less than or equal to one. The declaraion of **thread** specification is shown below:

$ThreadDecl \longrightarrow$ (**thread** $[Claim]$ $Block$ $[$**thread**$]$)
$Claim \longrightarrow$ **claim** $PermMap$

For instance, thread t0 in listing 3.1 claims full permission on a, b and c. This allows the thread to access those fields.

The permission held by threads can change over time. There are two ways that threads can gain or lose permission. The first way is by methods annotated with permission transfer specifications. The second way is by locks annotated with permission transfer specifications.

(**class** Math

public proc Add()
takes a@0.5, b@0.5, c@1.0

```
    post c' = a + b
    gives a@0.5, b@0.5, c@1.0

    obj a : int32 := 0
    obj b : int32 := 0
    obj c : int32 := 0


    (thread (*t0*) claim a@1.0, b@1.0, c@1.0
        //t0 requires access to a,b, c
        //and also transfers read permissions to t1
        //t0 receives the required permissions by claim specification
        b := 0
        c := 1
        Add()
     thread)

    (thread (*t1*)
        (while true
            (accept Add()
                c := a + b
            accept)
        while)
    thread)
class)
```

Listing 3.1: A program that illustrates permissions

3.4 Methods

A method can be annotated by permission specifications. The permission required by a method is declared by a **takes** annotation and the returned permission is declared by a **gives** annotation. Upon a call to a method that has a **takes** annotation, the caller loses the specified permission and the thread being called receives it. After the call, the callee loses the specified permission and the caller will receive it. To have a simpler specification, we can use **borrows** keyword instead of **takes** and **gives**

annotations with same permission map. The declaration of permission specifications in a method is as follows:

$$\begin{aligned} PermissionSpec \longrightarrow\ & \textbf{takes}\ PermMap \\ \mid\ & \textbf{gives}\ PermMap \\ \mid\ & \textbf{borrows}\ PermMap \end{aligned}$$

For instance, the method Add in lising 3.1 requires read permission on a and b which is specified by **takes**. Furthermore, it requires write permission on field c. The caller is responsible to provide permissions on these locations and they will be transferred from the caller to the callee. After execution, the method returns all the permission it received initially. This is specified by a **gives** annotation, which transfers the permissions from the callee to the caller.

In Add method, the chosen amount of read permission is 0.5 but it makes no difference what positive amount between zero and one is chosen. The fact is that choosing concrete read permissions for methods is a tedious task and it is preferred to do this in a more abstract way.

3.5 Abstract Read Permissions

Fractional permissions allow arbitrary many read permissions. A thread with a fraction of permission can split it many times, giving other threads read permission. However specifying the precise amount of permission for each method is a tedious task for the user. The fact is that the user usually needs only to distinguish between write or read permission and knowing the concrete value of the permission is mostly irrelevant. Moreover, the specification written with concrete permissions is

less reusable and it is more desirable to think about read permissions in more abstract way [24].

The main problem with concrete permission fractions is the reusability of specifications. Specifying a method with a concrete amount only allows the callers with equal or more permission to call the method. Method Add in listing 3.1 requires exactly 0.5 permission which only allows the callers with 0.5 or more permission to call the method. The fact is that a caller with read permission must be allowed to call any method with read permissions. This makes writing specification complicated because the permission of other threads must be considered too. Moreover, this makes the specifications dependent to implementation. It means that the changes in implementation of other threads may lead to rewriting the permission specification of the method.

```
(class Math

    public proc Add(ghost in ga, gb: Perm)
    takes a@ga, b@gb, c@1.0
    pre 0 < ga < 1 /\ 0 < gb < 1
    post c' = a + b
    gives a@ga, b@gb, c@1.0

    obj a : int32 := 0
    obj b : int32 := 0
    obj c : int32 := 0

    (thread (*t0*) claim a@1.0, b@1.0, c@1.0
        b := 0
        c := 1
        Add(0.5, 0.5)
    thread)

    (thread (*t1*)
        (while true
            (accept Add(in ga, gb : Perm)
```

```
            c := a + b
        accept)
    while)
thread)
class)
```

Listing 3.2: A program that illustrates abstract permission

To overcome the issues with concrete factional permissions, a more abstract way is implemented by ghost parameters. In this model, the caller decides the amount of given read permission and sends it via ghost parameters. This makes the callee independent of any specific amount of permission. This approach solves the problem of reusability and provides a more flexible reasoning method. To illustrate this, the example of listing 3.1 is rewritten using this approach. The thread t1 requires read permission on fields a and b. As shown in listing 3.2, the caller decides to give half of its permission on fields a and b. The method Add can be verified despite the exact amount of permission not being known.

3.6 Permission Transfer Scenarios

A thread can have a number of accept commands. In the previous examples, all the code is implemented by one method which takes permissions in the entry of the method and returns all of them in the existing. However, to have a higher level of concurrency, it is better to use two different methods to start an operation and finish it. This allows the client to execute concurrently with the started server thread. This structure complicates reasoning about the code. First, the client which started the operation may not be the one which finishes it. Second, the thread may not necessarily return the same permission it received. For example, it may call others

methods leading to gain or loss of some permission.

It is important to be sure that our model can handle these complications. In this section, we look at three representative patterns of communication and permission transfer between threads. Generally, a thread may start another thread and gives it some permission. For finishing the started thread, two cases are possible: the thread which started it will finish it (scenario 1) or another thread will finish it (scenario 2, 3). Verifying these example patterns ensures us that our chosen methodology can verify those and similar examples. In the next section, the three representative patterns will be discussed. In all of scenarios, there is a main client thread and two server threads. The server threads use two methods for synchronization, one for taking permission and the second one for giving permission. The first scenario simulate a method call in sequential programming except the start and the end of code is accessed by two calls. This feature increases parallelism. The second and third scenarios are useful for message passing where a thread sends a messages to other thread and finally receives back the message via another thread.

3.6.0 Scenario 1

In this scenario, the client thread sends a start message to one of server threads and gives it access permission. The started server thread starts the other server thread and gives its required access permission. Finally, both threads will be finished by the threads which have started them and they will return exactly the permissions they received initially.

To illustrate this scenario, an example in listing 3.3 is provided. There is one

client thread, t0_client and two worker threads, t1_server1 and t2_server2. Thread t0_client initially has full permission on field a. It starts another thread t1_server1 by calling worker1_start which requires read permission on field a. The caller is responsible to provide the required permission and gives half of its own permission to t1_server1. In order to have abstract permissions, the given permission is sent by ghost parameter p1, and it is asserted in precondition that it is a non-zero and partial permission. However, thread t1_server1 also needs this value out of the scope of worker1_start method because, for each memory access, it must be asserted that the thread has enough permission. Therefore, the initial amount of received permission is stored in a ghost field, gp1, and it is asserted in the postcondition that it is equal to the input permission parameter. The ghost field gp1 is added for reasoning and it must be guaranteed that it will not be modified until the worker1_finish block. For this reason, the thread makes gp1 read-only after storing the received permission. This is implemented by splitting the permission of field a by thread t1_server1 and transferring half of permission on the gp1 to the caller thread.

The started thread (t1_server1) also starts another thread which requires read permission too. Thread t1_server1 provides it by giving half of its read permission. Thread t2_server2 keeps all received permission and has no permission transfer while executing. Finally, thread t1_server1 which started the thread t2_server2 finishes it by calling worker2_finish. This makes transferring of permission to the caller which makes thread t1_server1 to retain its lost permission and have permission equal to gp1 again.

Finally, thread t0_client, which started t1_server1, will finish it by calling worker1_finish. The caller must return its permission on the ghost field gp1 which is asserted in the

precondition. Also, the callee returns its received permission in the postcondition of the method. However, the permission is stored in gp1 which cannot be mentioned in the postcondition anymore because the caller gave back its permission on gp1 in the precondition. For this reason, a copy of gp1 is sent by the ghost input parameter p1_init and it is asserted that p1_init is equal to gp1 in the precondition. Based on the fact that input parameters are not changeable, we can use p1_init in the postcondition to specify the returning permission. The caller can assume that all the given permission is returned.

```
(class C

    constructor()

    public proc main()

    //receives read permission on the field a and stores the amount in gp1
    public proc worker1_start(in ghost p1 : Perm)
    takes a@p1
    pre 0 < p1 /\ p1 < 1.0
    post gp1' == p1
    gives gp1@0.5

    //returns read permission equal to recieved permission via worker1_start
    public proc worker1_finish(in ghost p1_init : Perm)
    takes gp1@0.5
    pre p1_init == gp1
    gives a@p1_init

    //receives read permission on the field a and stores the amount in gp2
    public proc worker2_start(in ghost p2 : Perm)
    takes a@p2
    pre 0 < p2 /\ p2 < 1.0
    post gp2' == p2
    gives gp2@0.5

    //returns read permission equal to recieved permission via worker2_start
    public proc worker2_finish(in ghost p2_init : Perm)
```

```
takes gp2@0.5
pre p2_init == gp2
gives a@p2_init

public obj a : bool := 0
public ghost obj gp1, gp2 : Perm := 0

//t0 has full permission on the field a
//gives read permission to t1 and finally receives back it from t1
(thread (*t0_client*) claim a@1.0
    (while true
        invariant acc a@1.0
        (accept main()
            obj ghost pc1 : Perm := 0.5
            worker1_start(pc1)
            worker1_finish(pc1)
        accept)
    while)
thread)

(thread (*t1_server1*) claim gp1@1.0
    (while true
        invariant acc gp1@1.0
        (accept worker1_start(in ghost p1 : Perm)
            gp1 := p1
        accept)
        obj ghost pc2 : Perm := gp1 / 2
        worker2_start(pc2)
        worker2_finish(pc2)
        (accept worker1_finish(in ghost p1_init : Perm)
        accept)
    while)
thread)

(thread (*t2_server2*) claim gp2@1.0
    (while true
        invariant acc gp2@1.0
        (accept worker2_start(in ghost p2 : Perm)
            gp2 := p2
        accept)
        (accept worker2_finish(in ghost p2_init : Perm)
        accept)
```

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

Listing 3.3: A program that shows scenario 1 of permission transfer

3.6.1 Scenario 2

In this scenario, the client thread sends a start message to a server thread and gives it read permission. The started server thread sends a start message to another server thread and transfers it read permission too. Finally, the client thread finishes both of the server threads.

To illustrate this scenario, the example in listing 3.4 is provided. The thread t0_client starts server thread t1_server1 and gives it read permission on field a. Thread t1_server1 starts thread t2_server2 and gives it read permission on a too. This makes thread t1_server1 lose permission. However, it will not finish the child thread that it has started. As a result, it will finish with less permission that it was started with. Finally, the client thread t0_client will call the finish methods of both of t1_server1 and t2_server2. The reasoning about this scenario is similar to scenario 1 but two important issues must be declared in the specification. First, thread t1_server1 must acknowledge its caller about the given permission to thread t2_server2 enabling the caller to finish a thread which it has not started. Second, thread t1_server1 does not return the same permission it received and it must declare its new permission to its caller at the time of finishing.

As shown in the code, first, the server thread t1_server1 informs the client about its given permission to thread t2_server2. Thread t1_server1 stores the given permission

to thread t2_server2 in the ghost field gp2 and passes all of its permission on gp2 to the client because it does not need to access gp2 anymore and the client is responsible for finishing t2_server2. Also, to inform the client about the value of gp2, a specification is added in the postcondition of worker1_start stating the relation between the initial received permission (gp1) and the given permission (gp2). These two specifications enable the client thread to reason about a server thread that it has not started.

Second, thread t1_server1 must return its final permission which is less than what it received initially. It sends its final permission via the output parameter po1 and it is asserted that po1 is half of the initial received permission. Finally, thread t0_client finishes thread t1_server1 and receives half of the initial transferred permission. It also finishes thread t2_server2 and gains its returning permission. It can reason about thread t2_server2 because it was informed about it before. At this point, the main client will have received all given permission back and has full access on field a.

```
(class C

    constructor()

    public proc main()

    //receives read permission on the field a and stores the amount in gp1
    //gives read permission to t2
    public proc worker1_start(in ghost p1 : Perm)
    takes a@p1
    pre 0 < p1 and p1 < 1.0
    post gp1' == p1
    post gp2' == p1 / 2
    gives gp1@0.5, gp2@0.5

    //returns read permission to the caller
    //the retuned permission is the half of the permission
    //that it received via worker1_start
    public proc worker1_finish(in ghost p1_init : Perm,
```

```
                out po1 : Perm)
takes gp1@0.5
pre p1_init == gp1
post po1' == p1_init / 2
gives a@po1

//receives read permission on the field a and stores the amount in gp2
public proc worker2_start(in ghost p2 : Perm)
takes a@p2
pre 0 < p2 and p2 < 1.0
post gp2' == p2
gives gp2@0.5

//returns read permission to the caller
//the retuned permission is equal to the permission it received via worker2_start
public proc worker2_finish(in ghost p2_init : Perm,
                out ghost po2 : Perm)
takes gp2@0.5
pre p2_init == gp2
post p02' == p2_init
gives a@po2

public obj a : bool := 0
public ghost obj gp1, gp2 : Perm := 0

(thread (*t0_client*) claim a@1.0
    (while true
        (accept main()
            obj ghost p : Perm := 0
            obj ghost pc1, pr1, pr2 : Perm := 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
        obj ghost pc2 : Perm
        (accept worker1_start(in ghost p1 : Perm)
            gp1 := p1
            pc2 := gp1/2
            worker2_start(pc2)
        accept)
        (accept worker1_finish(in ghost p1_init : Perm,
                               out ghost po1 : Perm)
            po1 := gp1/2
        accept)
    while)
thread)

(thread (*t2_server2*) claim gp2@1
    (while true
        (accept worker2_start(in ghost p2 : Perm)
            gp2 := p2
        accept)
        (accept worker2_finish(in ghost p2_init : Perm,
                               out ghost po2 : Perm)
            po2 := p2_init
        accept)
    while)
thread)
class)
```

Listing 3.4: A program that shows scenario 2 of permission transfer

3.6.2 Scenario 3

In this scenario, the client thread starts both server threads and gives them read permission on field a. The first child thread must finish the other child thread even though it did not start the other child. Finally, the client will finish the remaining server thread.

An example of this scenario is provided in the listing 3.5. In this case, both

worker threads are started by thread t0_client. The server thread t2_server2 finishes thread t1_server1 and receives its permission. The client calls worker2_finish and will receive all the permission back. First, the child thread should be informed of the permission of the other thread to be able to reason about it. Second, it gains some permission by finishing the other thread and it must inform the main client about its new permission.

To handle this situation, server thread t2_server2 requires read permission on gp1 which will be given to it by the client t0_client. This helps it to reason about thread t1_server1. The returning permission of thread t2_server2 is the sum of its initial permission and thread t1_server1's initial permission. This is specified in its postcondition and it informs the main client by an output parameter.

```
(class C

    constructor()

    public proc main()

    //receives read permission on the field a and stores the amount in gp1
    public proc worker1_start(in ghost p1 : Perm)
    takes a@p1
    pre 0 < p1 and p1 < 1.0
    post gp1' == p1
    gives gp1@0.5

    //returns read permission equal to recieved permission via worker1_start
    public proc worker1_finish(in ghost p1_init : Perm)
    takes gp1@0.5
    pre p1_init == gp1
    gives a@p1_init

    //receives read permission on the field a and stores the amount in gp2
    public proc worker2_start(in ghost p2 : Perm)
    takes a@p2
    pre 0 < p2 and p2 < 1.0
```

```
post gp2' == p2
gives gp2@0.5

//receives read permission on gp1 and gp2
//which have the recieved permissions of t1 and t2
//returns the sum of its received read permission and the read permission of t1
public proc worker2_finish(in ghost p1_init, p2_init : Perm,
                  out ghost po : Perm)
takes gp1@0.5, gp2@0.5
pre p2_init == gp2
pre p1_init == gp1
post po' == p1_init + p2_init
gives a@po

public obj a : bool := 0
public ghost obj gp1, gp2 : Perm := 0.0

(thread (*t0_client*) claims a@1.0
    (while true
        invariant acc a@1.0
        (accept main()
            obj ghost p : Perm := 1.0
            obj ghost pc1, pc2, po : Perm := 0.0
            pc1 := p / 2
            p := p - pc1
            pc2 := p / 2
            p := p - pc2

            worker1_start(pc1)
            worker2_start(pc2)
            worker2_finish(pc1, pc2, po)
            p := p + po
        accept)
    while)
thread)

(thread (*t1_server1*) claim gp1@1.0
    (while true
        invariant acc gp1@1.0
        (accept worker1_start(in ghost p1 : Perm)
            gp1 := p1
        accept)
```

```
            (accept worker1_finish(in ghost p1_init : Perm)
            accept)
        while)
    thread)

    (thread (*t2_server2*) claim gp2@1.0
        (while true
            invariant acc gp2@1.0
            (accept worker2_start(in ghost p2 : Perm)
               gp2 := p2
            accept)
            (accept worker2_finish(in ghost p1_init, p2_init : Perm,
                                   out ghost po : Perm)
               worker1_finish(p1_init)
               po := p1_init + p2_init
            accept)
        while)
    thread)
class)
```

Listing 3.5: A program that shows scenario 3 of permission transfer

3.7 Locks and Class Invariants

The second way to synchronize accessing shared memory locations is locks. In the previous sections, threads were holders of access permissions and they transfer permissions between each other via the rendezvous mechanism. However, sometimes multiple threads must access a shared data concurrently and they must obtain and relinquish access while they are executing. To avoid data races, such accesses are protected by locks. In fact, lock objects become the holders of permissions. A thread must acquire the lock object to access shared data exclusively. The code can explicitly be annotated by **takes** and **gives**. A **takes** means the permission will explicitly transfer from the object to the thread after acquiring the lock. A **gives** will transfer

permission from the thread to the object after releasing the lock. However, often there is no need for permission transfer between the object and the thread. To check that a thread has sufficient permission for each data access inside the lock, it will be checked that the sum of the permissions of the thread and the lock allows the data access. Furthermore, a lock may have a guard section which requires permission to be checked. The syntax of **with** satements is as follows:

$Command \rightarrow$ (**with** Exp [**takes** $PermMap$] [$Guard$] [**do**]
$Block$ [**gives** $PermMap$] [**with**])

In this synchronization model, each class may be annotated by a **claim** specification which gives the specified permission to the object at initialization time. Also, each is associated with an invariant. The invariant specifies the permission which is held by the object. It may also have some conditions on the variables. The invariant must be valid after initialization and after each release of the mutual exclusion lock. The syntax of class is as follows:

$ClassDecl \longrightarrow$ (**class** [$ClassSpec$] $ClassMembers$ **class**)
$ClassSpec \longrightarrow Claim \mid Invariant$
$Invariant \longrightarrow$ **invariant** $Conditions$

For instance, class Counter in listing 3.6 has the shared field count. It is claimed with half permission by the object of the class and so the invariant is valid after initialization.

(**class** Counter

claim count@0.5
invariant acc count@0.5 /\ count >= 0

constructor()

proc increment()

```
    takes count@0.5
    pre count >= 0
    post count' > 0
    gives count@0.5

    obj count : real32 := 0

    (thread (*t0*)
        (while true
            (accept increment()
                (with this
                        count := count + 1
                with)
            accept)
        while)
    thread)
class)
```

Listing 3.6: A program that shows class invariant and a lock block

Method increment modifies field count and needs full access. In the precondition, it takes half permission. After successful acquisition of the lock, it is allowed to modify field count. In fact, for each data access inside the lock, it is checked whether the sum of the permission of the thread and the lock allows the access. The thread releases the lock after modification. The invariant must hold at the time the lock is released, which it does for this example.

3.8 Parallelism

The **co** (concurrent) statement declares that two or more blocks of code execute in parallel. Each part of a **co** command executes independently and they may access shared data concurrently. To avoid data races, it must be checked that the **co** threads have write access to disjoint locations. For this reason, each **co** thread explicitly

specifies its required permission on locations by a **claim** specification. Their required permissions will be transferred from the parent to the children. As a result, the sum of permissions required by **co** threads cannot exceed the permission of the parent thread. This rules out the possibility of data races. At the end, the **co** threads will return their permissions to the parent thread. The syntax of **co** statement is declared as follows:

$$Command \longrightarrow (\mathbf{co}\ \underline{[}Claim\underline{]}\ Block\ \underline{(}\ ||\ \underline{[}Claim\underline{]}\ Block\underline{)}^{+}\ \underline{[}\mathbf{co}\underline{]})$$

The following example in listing 3.7, thread t0 has **co** statements. Both **co** blocks require read access on location a and write accesses on disjoint locations. The parent thread will transfer the required permission to **co** threads. We can see that it will be an error if both threads request for write permission on same location because the permission cannot be duplicated. Finally, the **co** threads return all the initial received permission to the parent thread.

```
(class C

    constructor()

    public proc Add()
    takes a
    pre a >= 0
    pre lsint32(a+2)
    post a' = a + 2
    gives a

    obj a : int32 := 0

    (thread (*to*)
        (while true
            (accept Add()
                obj sum1, sum2 : int32 := 0
                (co
                    claims a@0.5, sum1@1.0
```

```
                sum1 := a + 1
            ||
                claims a@0.25, sum2@1.0
                sum2 := a + 1
            co)
            a := sum1 + sum2
        accept)
    while)
thread)
class)
```

Listing 3.7: A program that illustrates co statement specifications

3.9 Loop Invariant

A loop may contain infinite iterations which makes it impossible to verify all of them individually. For this reason, loops are specified by loop invariants. The invariant must hold at the entry to the loop. Also, it must be valid at the end of each iteration of the loop. The chosen loop invariant must be strong enough to prove the post-condition of the method too. The invariant consists of access permissions annotated with keyword **acc** and condition specifications. The declarations of loop invariant for **while** and **for** satements are shown below:

$$\begin{aligned} \mathit{Command} \longrightarrow\ & (\textbf{while}\ \mathit{Expression}\ \underline{[}\mathit{Invariant}\underline{]}\ \underline{[}\textbf{do}\underline{]}\ \mathit{Block}\ \underline{[}\textbf{while}\underline{]}) \\ & \mid (\textbf{for}\ \mathit{Name} : \mathit{Bounds}\ \underline{[}\mathit{Invariant}\underline{]}\ \underline{[}\textbf{do}\underline{]}\ \mathit{Block}\ \underline{[}\textbf{for}\underline{]}) \end{aligned}$$

The example in listing 3.8 illustrates a loop invariant for **while** statement. The method calculates the square root of field n by while loop.

```
(class Math

    constructor()

    public proc Sqrt()
    takes n@1.0
```

```
    pre 0 <= n
    post n' * n' <= n && n' < (n+1) * (n+1)
    gives n@1.0

    obj n : int32 := 100

    (thread (*t0*)
        (while true
            (accept Sqrt()
                obj m : int32 := n
                n := 0
                (while ((n+1)*(n+1) <= m)
                    invariant acc n@1.0 && n*n <= m
                    n := n + 1
                        while)
                accept)
            while)
    thread)
class)
```

Listing 3.8: A program that shows while statement with loop invariant

The example in listing 3.9 is provided to show a **for** loop with an invariant. The Add method adds input parameter n to the field sum by **for** loop.

```
(class Math

    constructor()

    procedure Add(in n : int32)
    takes sum@1.0
    pre n >= 0 && sum >= 0
    pre lsint32(sum+n)
    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)
```

Listing 3.9: A program that shows for statement with loop invariant

3.10 Array

HARPO/L supports array objects. In reasoning about arrays, we only consider permission on the level of items of arrays with primitive types. This allows multiple threads to work concurrently on separate elements of an array. For instance, in the following example, the method takes write permission only on even elements of array and returns the permission in those locations at the end.

```
(class Math
    proc Even_Op()
    takes {i : {0,..n} when even(i) do a(i)}@1.0
    gives {i : {0,..n} when even(i) do a(i)}@1.0

    const n := 10
    obj a : int32[n] := (for i : n do 0 for)

    (thread (*t0*)
        (while true
            (accept Even_Op()
                //implementation code
            accept)
        while)
    thread)
class)
```

Listing 3.10: A program that illustrates arrays

Chapter 4

Translating HARPO/L to Boogie

This chapter presents translation of the HARPO programming language and the specification constructs to Boogie based on our methodology. The translation into Boogie consists of two steps. First, a prelude is declared which encodes some properties of all HARPO/L programs. Then, the specific program is translated into Boogie.

Note that ghost variables are behaved as regular variables in Boogie. For this reason, **ghost** keyword is ignored in the translation.

4.0 Prelude

The translation from HARPO/L to Boogie starts with a prelude. The prelude is independent of the program being translated. In fact, it encodes some properties that are required for translation of all HARPO/L programs. The final Boogie program consists of the prelude and the translation of the specific program.

4.0.0 Modeling Memory

One of the first and important decisions when designing a translation from a source language into Boogie is the modeling of the memory. HARPO/L is an object oriented language with object references and fields. The memory can be represented in several models. We chose to define the heap (memory) as a variable which maps object references and fields to values.

type *Ref*;
type *Field* α;
type *HeapType* $= \langle\alpha\rangle[Ref, Field\ \alpha]\alpha$;
var *Heap* : *HeapType*;

The translation into Boogie introduces a type constructor *Ref* to model object references. All the reference types in HARPO/L are mapped to Boogie type *Ref*. The memory is represented by global variable *Heap*. Its type is a polymorphic map because the fields can have different values depending on the field name used. Each field declared in HARPO/L gives rise to a unique value of type *Field* α, where α is the type of the field.

Furthermore, to model arrays another heap is required. To store the elements of arrays, a global variable, *ArrayHeap*, is defined. *ArrayHeap* maps array references and indices to the value of elements.

type *ArrayRef* α;
type *ArrayHeapType* $= \langle\alpha\rangle[ArrayRef\ \alpha, int]\alpha$;
var *ArrayHeap* : *ArrayHeapType*;

4.0.1 Reference Types

To model class and interface types, the prelude declares a Boogie type:

type $ClassName$;

Every class declaration gives rise to a particular value of *ClassName* type. Because all HARPO/L objects are modelled by values of type *Ref*, a function *dtype* is added to the translation. Function dtype returns the run-time type of a reference.

function $dtype(Ref)$ **returns** $(ClassName)$;

4.0.2 Type Axioms

HARPO/L supports several bounded integer types with different sizes. All of HARPO/L integer types are translated to **int** type in Boogie. To check that integer types are in their associated bound, we declare a function and axiom for each integer type:

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 <= $max64$);

Furthermore, the following function is added to check the range of integers. (The function **Tr** translates a HARPO/L program to a Boogie program.)

$$
\begin{aligned}
&Isint(E.f)_{H1,H2,P} \equiv \\
&\quad ((Type(E.f) == \textbf{int8}) ==> Isint8(H1[Tr[E]_{H1,H2,P}, C.f]) \;||\\
&\quad (Type(E.f) == \textbf{int16}) ==> Isint16(H1[Tr[E]_{H1,H2,P}, C.f]) \;||\\
&\quad (Type(E.f) == \textbf{int32}) ==> Isint32(H1[Tr[E]_{H1,H2,P}, C.f]) \;||\\
&\quad (Type(E.f) == \textbf{int64}) ==> Isint64(H1[Tr[E]_{H1,H2,P}, C.f]));
\end{aligned}
$$

$$
\begin{aligned}
&Isint(E[i])_{H1,H2,P} \equiv \\
&\quad ((Type(E) == \textbf{int8}) ==> Isint8(ArrayH1[Tr[E]_{H1,H2,P}, Tr[i]_{H1,H2,P}]) \;||\\
&\quad (Type(E) == \textbf{int16}) ==> Isint16(ArrayH1[Tr[E]_{H1,H2,P}, Tr[i]_{H1,H2,P}]) \;||\\
&\quad (Type(E) == \textbf{int32}) ==> Isint32(ArrayH1[Tr[E]_{H1,H2,P}, Tr[i]_{H1,H2,P}]) \;||\\
&\quad (Type(E) == \textbf{int64}) ==> Isint64(ArrayH1[Tr[E]_{H1,H2,P}, Tr[i]_{H1,H2,P}]));
\end{aligned}
$$

4.0.3 Array Length

The length of an array is modeled with the *Length* function. For each array declaration, we can add an axiom regarding its length.

function *Length* $\langle x \rangle$(*Field* (*ArrayRef* x)) **returns** (**int**);

4.0.4 Permission

A permission is a real number between zero and one. The translation introduces type *Perm* to model permissions.The type *Perm* is a type synonym of **real** type. Boogie

handles floating-point approximation and it is guaranteed that no permission loss will happen. Furthermore, the permissions of all fields are encoded as a map from locations to the permissions on those locations. A new type *PermissionType* will be declared in the prelude:

type $Perm = \mathbf{real}$;
type $PermissionType : \langle\alpha\rangle[Ref,\ Field\ \alpha]Perm$;

To keep track of permission, each thread has its own local variable *Permission* of type *PermissionType*. A permission p is called "full permission" if p is equal to 1.0. It is called "some permission" if p is greater than zero. Furthermore, it is called "no permission" if p is equal to zero. We introduce three shorthands for the common permission requirements. *CanRead* checks having some permission, *CanWrite* checks having full permission and *CanAccess* checks having the specified permission:

$CanRead(E.f)_{H1,H2,P} \equiv P[Tr[E]_{H1,H2,P}, C.f] > 0.0$
$CanWrite(E.f)_{H1,H2,P} \equiv P[Tr[E]_{H1,H2,P}, C.f] == 1.0$
$CanAccess(E.f@n)_{H1,H2,P} \equiv P[Tr[E]_{H1,H2,P}, C.f] == n$
$CanAccess(E.f)_{H1,H2,P} \equiv P[Tr[E]_{H1,H2,P}, C.f] == 1.0$

Pemissions can be incremented or decremented. Two operations are introduced to model permission changes. *AddPermission* adds the declared permission to the current permission. *RemovePermission* checks that the current thread holds enough permission and then decreases it. These operations are declared as follows:

$AddPermission(E.f@n)_{H1,H2,P} \equiv$
    $P[Tr[E]_{H1,H2,P}, C.f] = P[Tr[E]_{H1,H2,P}, C.f] + n$
$AddPermission(E.f)_{H1,H2,P} \equiv$
    $P[Tr[E]_{H1,H2,P}, C.f] = P[Tr[E]_{H1,H2,P}, C.f] + 1.0$
$RemovePermission(E.f@n)_{H1,H2,P} \equiv$
    assert $P[Tr[E]_{H1,H2,P}, C.f] >= n$
    $P[Tr[E]_{H1,H2,P}, C.f] = P[Tr[E]_{H1,H2,P}, C.f] - n$
$RemovePermission(E.f)_{H1,H2,P} \equiv$

assert $P[Tr[E]_{H1,H2,P}, C.f] == 1.0$
$P[Tr[E]_{H1,H2,P}, C.f] = P[Tr[E]_{H1,H2,P}, C.f] - 1.0$

Additionally, to track permission on array elements, permission type *ArrayPermissionType* is declared:

type $ArrayPermissionType : \langle\alpha\rangle[ArrayRef\ \alpha, \mathbf{int}]Perm;$

Each thread declares a new local variable, *ArrayPermission* to store its permission on elements of arrays. Permission shorthands for arrays are declared as follows:

$CanRead(E.f[i])_{H1,H2,P} \equiv P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] > 0.0$
$CanWrite(E.f[i])_{H1,H2,P} \equiv P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] == 1.0$
$CanAccess(E.f[i]@n)_{H1,H2,P} \equiv P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] == n$
$CanAccess(E.f[i])_{H1,H2,P} \equiv P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] == 1.0$

In the following part, permission operations for arrays are declared:

$AddPermission(E.f[i]@n)_{H1,H2,P} \equiv$
$P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] = P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] + n;$

$AddPermission(E.f[i])_{H1,H2,P} \equiv$
$P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] = P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}]+1.0;$

$AddPermission(\{i : Set\ \mathbf{when}\ Cond\ \mathbf{do}\ E\}@n)_{H1,H2,P} \equiv$
assert $Df[E]_{H1,H2,P}\ /\backslash\ Df[Set]_{H1,H2,P}\ /\backslash\ Df[Cond]_{H1,H2,P};$
foreach i **in** $Tr[Set]_{H1,H2,P};$
if $Tr[Cond]_{H1,H2,P}\ \{AddPermission(E@n)_{H1,H2,P};\}$

$AddPermission(\{i : Set\ \mathbf{when}\ Cond\ \mathbf{do}\ E\})_{H1,H2,P} \equiv$
assert $Df[E]_{H1,H2,P}\ /\backslash\ Df[Set]_{H1,H2,P}\ /\backslash\ Df[Cond]_{H1,H2,P};$
foreach i **in** $Tr[Set]_{H1,H2,P}$
if $Tr[Cond]_{H1,H2,P}\ \{AddPermission(E@1.0)_{H1,H2,P};\}$

$RemovePermission(E.f[i]@n)_{H1,H2,P} \equiv$
assert $P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] >= n;$
$P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] = P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] - n;$

$RemovePermission(E.f)_{H1,H2,P} \equiv$
assert $P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] == 1.0;$

$$P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] = P[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] - 1.0;$$

$RemovePermission(\{i : Set \textbf{ when } Cond \textbf{ do } E\}@n)_{H1,H2,P} \equiv$
  assert $Df[E]_{H1,H2,P} \ /\backslash\ Df[Set]_{H1,H2,P} \ /\backslash\ Df[Cond]_{H1,H2,P}$;
  foreach i **in** $Tr[Set]_{H1,H2,P}$
    if $Tr[Cond]_{H1,H2,P}$ $\{RemovePermission(E@n)_{H1,H2,P};\}$

$RemovePermission(\{i : Set \textbf{ when } Cond \textbf{ do } E\})_{H1,H2,P} =$
  assert $Df[E]_{H1,H2,P} \ /\backslash\ Df[Set]_{H1,H2,P} \ /\backslash\ Df[Cond]_{H1,H2,P}$;
  foreach i **in** $Tr[Set]_{H1,H2,P}$
    if $Tr[Cond]_{H1,H2,P}$ $\{RemovePermission(E@1.0)_{H1,H2,P};\}$

Generally, when adding permission to a thread via claims or any synchronization method, if the thread has zero permission on a location, we havoc that location to an arbitrary value. The reason is that other threads may have changed the value of that location. The operation *HavocNewLoc* is added for this purpose; it will be used when adding permission. First, it checks whether the permission on the location is zero. Then, it havocs that location in the heap.

$HavocNewLoc(E.f)_{H1,H2,P} \equiv$
  if $(P[Tr[E]_{H1,H2,P}, C.f] == 0.0)$
    $\{$**havoc** H_tmp;
      $H1[Tr[E]_{H1,H2,P}, C.f] := H\_tmp[Tr[E]_{H1,H2,P}, C.f];\}$

$HavocNewLoc(E.f[i])_{H1,H2,P} \equiv$
  if $(ArrayP[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] == 0.0)$
    $\{$**havoc** $ArrayH\_tmp$;
      $ArrayH1[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}] :=$
        $ArrayH\_tmp[H1[Tr[E]_{H1,H2,P}, C.f], Tr[i]_{H1,H2,P}];\}$

$HavocNewLoc(\{i : Set\ when\ Cond\ do\ E\})_{H1,H2,P} \equiv$
  foreach i **in** $Tr[Set]_{H1,H2,P}$;
    if $Tr[Cond]_{H1,H2,P}$ $\{HavocNewLoc(E)_{H1,H2,P};\}$

4.1 Translation

A HARPO/L program consists of a set of class, interface and object declarations:

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

To translate programs, we define the function **Tr**. It takes a HARPO/L program as input and generates a Boogie program. In the following section, the translation of a program will be described. C is used to represent current class and B is used to represent current thread.

The code in listing 4.0 shows a HARPO/L program. It has two threads. The thread $t0$ implements a method and the thread $t1$ calls the method. In the next sections, we will use this code to show the translation of HARPO/L constructs.

```
(class Counter

    constructor()

    public proc increment(in n : int32)
    takes count
    pre count >= 0 && n > 0
    pre lsint32(count+n)
    post count' == count + n
    gives count

    obj count : int32 := 0

    (thread (*t0*)
        (while true
            (accept increment(in n : int32)
                count := count + n
            accept)
        while)
    thread)

    (thread (*t1*) claim count
        count := 0
```

```
        increment(1)
        assert count == 1
    thread)
class)
```

Listing 4.0: A HARPO/L program with specifications

4.1.0 Classes

Each class is translated to a constant in Boogie program.

$Tr[(\textbf{class}\ C\ members\ \textbf{class})]_{H1,H2,P} =$
    const unique $C : ClassName$;
    $Tr^*[members]_{H1,H2,P}$;

the **unique** modifier declares that the value of the constant differs from other unique constants. Translation of members is denoted by $Tr^*[members]_{H1,H2,P}$ meaning function Tr is applied to all the members. For instance, the translation of class *Counter* in listing 4.0 is the following code:

const unique Counter : ClassName;

A class may implement one or more interfaces. To express subtyping in Boogie, $<:$ is used. For example, if class C implements interface J, the translation in Boogie is the following code:

$Tr[(\textbf{class}\ C\ \textbf{implements}\ J\ \ldots\ \textbf{class})]_{H1,H2,P} =$
    const unique $C : ClassName <: J$;

which means that J is the only direct parent of C.

4.1.1 Interfaces

An interface is translated to a constant in Boogie:

$Tr[(\textbf{interface}\ K\ members\ \textbf{interface})]_{H1,H2,P} =$
    const unique $K : ClassName$;

4.1.2 Fields

The translation of each field produces a unique *Field* value as follows:

$Tr[\textbf{obj}\ f : T := E]_{H1,H2,P} =$
    const unique $C.f : Field\ Tr[T]_{H1,H2,P}$;

Boogie allows using "." character in identifier names. The name of the class is prepended to the field name. For example, the *Counter* class has a field *count* which is translated to:

const unique Counter.count : Field **int**;

4.1.3 Constants

Constants are translated to constants in Boogie. The value of the constant is declared by an axiom.

$Tr[\textbf{const}\ x : T := E]_{H1,H2,P} =$
    const $x : Tr[T]_{H1,H2,P}$;
    axiom $Df[E]_{H1,H2,P}$ && $x == Tr[E]_{H1,H2,P}$;

4.1.4 Types

In HARPO/L, types are primitive types, references to objects, permission type and arrays. The primitive types include integers, reals and Boolean.

$Types \longrightarrow int8 \mid int16 \mid int32 \mid int64 \mid real16 \mid real32 \mid real64 \mid bool \mid CName \mid Perm \mid T[N]$

The translation of types is as follows:

$Tr[int8]_{H1,H2,P} = \textbf{int}$
$Tr[int16]_{H1,H2,P} = \textbf{int}$
$Tr[int32]_{H1,H2,P} = \textbf{int}$
$Tr[int64]_{H1,H2,P} = \textbf{int}$
$Tr[int]_{H1,H2,P} = \textbf{int}$
$Tr[real16]_{H1,H2,P} = \textbf{real}$
$Tr[real32]_{H1,H2,P} = \textbf{real}$
$Tr[real64]_{H1,H2,P} = \textbf{real}$
$Tr[real]_{H1,H2,P} = \textbf{real}$
$Tr[bool]_{H1,H2,P} = \textbf{bool}$
$Tr[CName]_{H1,H2,P} = Ref$ where $CName$ is a reference to an instance of a class
$Tr[Perm]_{H1,H2,P} = Perm$
$Tr[T[N]]_{H1,H2,P} = ArrayRef\ Tr[T]_{H1,H2,P}$

4.1.5 Expressions

HARPO/L expressions include arithmetic and logical operators. Translation is done by two functions. First, the well-definedness of an expression is checked by translation function *Df* . It generates a Boogie predicate which checks whether an expression is well defined in HARPO/L. For example, the expression x/y is defined if y is not zero. Second, a HARPO/L expression E is translated to an equivalent Boogie expression by *Tr*.

The *Tr* and *Df* functions are parameterized by two heap parameters $H1$ and $H2$ and one permission parameter P. Translation of expressions is based on $H1$ and translation of expressions with apostrophe is based on $H2$. In two state expressions (postconditions), both parameters must be provided. However, in single state expressions, the second heap, $H2$, can be blank.

$Df[E => F]_{H1,H2,P} = Df[E]_{H1,H2,P}\ /\backslash\ Df[F]_{H1,H2,P}$
$Df[E <= F]_{H1,H2,P} = Df[E]_{H1,H2,P}\ /\backslash\ Df[F]_{H1,H2,P}$

$Df[E <=> F]_{H1,H2,P} = Df[F]_{H1,H2,P} \ /\backslash\ Df[E]_{H1,H2,P}$
$Df[E \ /\backslash\ F]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P}$
$Df[E \ \backslash/\ F]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P}$
$Df[\tilde{}E]_{H1,H2,P} = Df[E]_{H1,H2,P}$
$Df[E \copyright F]_{H1,H2,P} =$
$\quad Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P}$ with $\copyright$ being $=$, $\tilde{}=$, $<$, $\_<$, $>\_$, $>$
$Df[E \ \oplus\ F]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P}$ with $\oplus$ being +,-,*
$Df[E \ / \ F]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P} \ /\backslash\ Tr[F]_{H1,H2,P} \neq 0$
$Df[E \ \mathbf{mod}\ F]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P} \ /\backslash\ Tr[F]_{H1,H2,P} \neq 0$
$Df[E.f]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash$
$\quad CanRead(E.f)_{H1,H2,P}$ where $E.f$'s type is primitive and not integer
$Df[E.f]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ CanRead(E.f)_{H1,H2,P} \ /\backslash$
$\quad Isint(E.f)_{H1,H2,P}$ where $E.f$'s type is integer
$Df[E.f]_{H1,H2,P} = Df[E]_{H1,H2,P}$ where $E.f$'s type is not primitive
$Df[E']_{H1,H2,P} = Df[E]_{H2,-,P}$
$Df[E[F]]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P} \ /\backslash$
$\quad 0 \leq Tr[F]_{H1,H2,P} \ /\backslash\ Tr[F]_{H1,H2,P} < length(Tr[E]_{H1,H2,P}) \ /\backslash$
$\quad CanRead(E[F])_{H1,H2,P}$ where $E[F]$'s type is primitive and not integer
$Df[E[F]]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P} \ /\backslash$
$\quad 0 \leq Tr[F]_{H1,H2,P} \ /\backslash\ Tr[F]_{H1,H2,P} < length(Tr[E]_{H1,H2,P}) \ /\backslash$
$\quad CanRead(E[F])_{H1,H2,P} \ /\backslash$
$\quad Isint(E.f)_{H1,H2,P}$ where $E[F]$'s type is integer
$Df[E[F]]_{H1,H2,P} = Df[E]_{H1,H2,P} \ /\backslash\ Df[F]_{H1,H2,P} \ /\backslash\ 0 \leq Tr[F]_{H1,H2,P} \ /\backslash$
$\quad Tr[F]_{H1,H2,P} < length(Tr[E]_{H1,H2,P})$ where $E[F]$'s type is not primitive

$Tr[E => F]_{H1,H2,P} = Tr[E]_{H1,H2,P} ==> Tr[F]_{H1,H2,P}$
$Tr[E <= F]_{H1,H2,P} = Tr[F]_{H1,H2,P} ==> Tr[E]_{H1,H2,P}$
$Tr[E <=> F]_{H1,H2,P} = Tr[E]_{H1,H2,P} <==> Tr[E]_{H1,H2,P}$
$Tr[E \ /\backslash\ F]_{H1,H2,P} = Tr[E]_{H1,H2,P} \ /\backslash\ Tr[F]_{H1,H2,P}$
$Tr[E \ \backslash/\ F]_{H1,H2,P} = Tr[E]_{H1,H2,P} \ \backslash/\ Tr[F]_{H1,H2,P}$
$Tr[\tilde{}E]_{H1,H2,P} = !Tr[E]_{H1,H2,P}$
$Tr[E \copyright F]_{H1,H2,P} =$
$\quad Tr[E]_{H1,H2,P} \copyright Tr[F]_{H1,H2,P}$ with $\copyright$ being $=$, $\tilde{}=$, $<$, $\_<$, $>\_$, $>$
$Tr[E \ \oplus\ F]_{H1,H2,P} = Tr[E]_{H1,H2,P} \ \oplus\ Tr[F]_{H1,H2,P}$ with $\oplus$ being +, -, *
$Tr[E \ / \ F]_{H1,H2,P} = Tr[E]_{H1,H2,P} \ / \ Tr[F]_{H1,H2,P}$
$Tr[E \ \mathbf{mod}\ F]_{H1,H2,P} = Tr[E]_{H1,H2,P} \ \mathbf{mod}\ Tr[F]_{H1,H2,P}$
$Tr[E.f]_{H1,H2,P} = H1[Tr[E]_{H1,H2,P}, C.f]$
$Tr[E']_{H1,H2,P} = Tr[E]_{H2,-,P}$
$Tr[E[F]]_{H1,H2,P} = ArrayH1[Tr[E]_{H1,H2,P}, Tr[F]_{H1,H2,P}]$

4.1.6 Statements

HARPO/L programs will have undergone some preprocessing prior to translation. We can assume that the input program being translated has following properties:

0. Prefix **this** has been added to all references to fields or methods in the scope of a class. For instance, if a class declares a field F, each reference of F has been changed to **this**.F in that class.

1. All local variables of a class are promoted to fields in that class.

Considering these properties of a program, the translation of statements is described in this section.

- **Local Variable Declaration**

Provided that local variables are converted as fields of the class prior to translation, the remaining of translation of local variable declaration, is shown in below:

$$
\begin{array}{l}
Tr[\textbf{obj } f : T := E \ block]_{H1,H2,P} = \\
\quad AddPermission(\textbf{this}.f)_{H1,H2,P}; \\
\quad \textbf{assert } Df[E]_{H1,H2,P}; \\
\quad H1[\textbf{this}, C.f] := Tr[E]_{H1,H2,P}; \\
\quad \textbf{assert } Isint(\textbf{this}.f)_{H1,H2,P}; \qquad \text{where } f \text{ is integer} \\
\quad Tr^*[block]_{H1,H2,P}; \\
\quad RemovePermission(\textbf{this}.f)_{H1,H2,P};
\end{array}
$$

First, full permission on variable f is added to the thread. Then the heap is updated by initial value of the variable after checking it is well-defined. A local variable is defined in its scope and it is not possible to access it out of its defined scope. As a result, after translation of the code in the scope of the variable, the thread loses permission on the variable.

- **Assignment**

Based on the fact that local variables are promoted to fields prior to translation, the only assignments are updating fields and elements of arrays. Additionally, HARPO/L does not allow updating fields with reference types. Consequently, the right-hand side of an assignment can only be expression with a primitive type.

Translation of assignment checks that all involved expressions are well-defined. Then, it checks that the thread has write permission on the field. Finally, it updates the heap based on the new value.

$Tr[E.f := F]_{H1,H2,P} =$
assert $Df[E]_{H1,H2,P}$;
assert $Df[F]_{H1,H2,P}$;
assert $CanWrite(E.f)_{H1,H2,P}$;
$H1[Tr[E]_{H1,H2,P}, C.f] := Tr[F]_{H1,H2,P}$;
assert $Isint(E.f)_{H1,H2,P}$; where $E.f$ is integer

Array update checks that all expressions are well-defined. Then it checks that the thread has write permission on the element and read permission on the index. The *ArrayHeap* is updated based on the new value.

$Tr[E[F] := G]_{H1,H2,P} =$
assert $Df[E[F]]_{H1,H2,P}$;
assert $Df[G]_{H1,H2,P}$;
assert $CanWrite(E[F])_{H1,H2,P}$;
$ArrayH1[Tr[E]_{H1,H2,P}, Tr[F]_{H1,H2,P}] := Tr[G]_{H1,H2,P}$;
assert $Isint(E[F])_{H1,H2,P}$; where $E[F]$ is integer

- **if Statement**

To translate an **if** statement, first it is checked that the guard is defined. Then it translates HARPO/L **if** statement into Boogie's **if** statement.

$Tr[(\textbf{if } (E) \textbf{ then } S0 \textbf{ else } S1 \textbf{ if})]_{H1,H2,P} =$
$\quad \textbf{assert } Df[E]_{H1,H2,P};$
$\quad \textbf{if } (Tr[E]_{H1,H2,P}) \{Tr^*[S0]_{H1,H2,P}\} \textbf{ else } \{Tr^*[S1]_{H1,H2,P}\};$

The example in listing 4.1 shows a HARPO/L program implemented with an **if** statement.

```
(class C

    constructor()

    proc m(in x:real32, out y:bool)
    takes n@0.5
    post x < n ==> y == 1
    post x >= n ==> y == 0
    gives n@0.5

    obj n : real32 := 100

        (thread (*t0*)
            (while true
                (accept m(in x:real, out y:bool)
                        (if x < n
                            y := 1
                        else
                            y := 0
                        if)
                accept)
            while)
    thread)
class)
```

Listing 4.1: A HARPO/L program that illustrates if statement

The translation of the **if** section of the example in listing 4.1 is shown below. First, it checks that the thread has enough permission on guard expression. Then it translates the **if** statement in HARPO/L to an **if** statement in Boogie.

```
assert Permission[this, C.x] > 0.0 && Permission[this, C.n] > 0.0;
```

```
if (Heap[this, C.x] < Heap[this, C.n])
    {assert Permission[this, C.y] == 1.0;
    Heap[this, C.y] := true;}
else
    {assert Permission[this, C.y] == 1.0;
    Heap[this, C.y] := false;}
```

- **While Statement**

To translate a **while** statement, the translation stores the value of the heaps in old heap variables *oldH* and *oldArrayH* on entering to the loop. Then it translates the **while** statement in HARPO/L to a **while** statement in Boogie. The loop invariant is translated to a loop invariant in Boogie and it is checked whether the invariant is defined. Additionally, it adds an invariant to assert the guard is well-defined. The encoding also adds an invariant to assert that the locations of the heap which are not modified by the loop hold their values. This invariant is generated based on the locations that the invariant does not specify full permission on them. The reason for this invariant is that Boogie havocs the modified variables by the loop and then assumes the invariant. Therefore, if an array variable such as heap is changed, the invariant must assert the unchanged locations too.

$Tr[(\textbf{while}\ E\ \textbf{invariant}\ J\ stmts\ \textbf{while})]_{H,sH,P} =$
$\quad oldH := H;$
$\quad oldArrayH := ArrayH;$
$\quad \textbf{while}\ (Tr[E]_{H,oldH,P})$
$\quad\quad \textbf{invariant}\ Df[J]_{H,oldH,P}\ \&\&\ Tr[J]_{H,oldH,P};$
$\quad\quad \textbf{invariant}\ Df[E]_{H,oldH,P};$
$\quad\quad \textbf{invariant}\ (\textbf{forall}\ \langle x\rangle\ r : Ref,\ f : Field\ x :: !(r.f\ \textbf{in}\ ModifiedVariable(J))$
$\quad\quad\quad ==> H[r,f] == oldH[r,f]);$
$\quad\quad \textbf{invariant}\ (\textbf{forall}\ \langle x\rangle\ r : ArrayRef\ x,\ f : int :: !(r[f]\ \textbf{in}\ ModifiedVariable(J))$
$\quad\quad\quad ==> ArrayH[r,f] == oldArrayH[r,f]);$
$\quad \{Tr^*[stmts]_{H,oldH,P}\}$

For instance, the method sqrt in listing 3.8 is implemented by **while** loop. The translation of **while** part is as follows:

```
oldHeap := Heap;
oldArrayHeap := ArrayHeap;
while ((Heap[this, Math.n]+1) * (Heap[this, Math.n]+1) <=
        Heap[this, Math.m])
    invariant Permission[this, Math.n] == 1.0 &&
        (Heap[this, Math.n]*Heap[this, Math.n] <= Heap[this, Math.m]);
    invariant (forall<x> r:Ref, f : Field x :: !(r == this && f == Math.n) ==>
        Heap[r, f] == oldHeap[r, f]);
{
    assert Permission[this, Math.n] == 1.0;
    assert Isint32(Heap[this, Math.n]);
    Heap[this, Math.n] := Heap[this, Math.n] + 1;
    assert Isint32(Heap[this, Math.n]);
}
```

- **For Statement**

The **for** loop is translated to a **while** loop in Boogie.

$Tr[(\textbf{for}\ x : E\ \textbf{invariant}\ J\ \textbf{do}\ stmts\ \textbf{for})]_{H,oldH,P} =$
$oldH := H;$
$oldArrayH := ArrayH;$
while $(x < Tr[E]_{H,oldH,P})$
invariant $Df[J]_{H,oldH,P}$ && $Tr[J]_{H,oldH,P}$;
invariant $Df[E]_{H,oldH,P}$;
invariant (**forall** $\langle x \rangle\ r : Ref,\ f : Field\ x$ ::!$(r.f$ **in** $ModifiedVariable(J))$
$==> H[r, f] == oldH[r, f]$);
invariant (**forall** $\langle x \rangle\ r : ArrayRef\ x,\ f : int$ ::!$(r[f]$ **in** $ModifiedVariable(J))$
$==> ArrayH[r, f] == oldArrayH[r, f]$);
{
$Tr^*[stmts]_{H,oldH,P}$;
$x := x + 1$;
}

For instance, the method Add in listing 3.9 is implemented with a **for** loop. The translation of the **for** loop is as follows:

```
oldHeap := Heap;
oldArrayHeap := ArrayHeap;
while(Heap[this, Math.i] < Heap[this, Math.n])
    invariant Permission[this, Math.i] == 1.0 && 0 <= Heap[this, Math.i]
        && Heap[this, Math.i] <= Heap[this, Math.n];
    invariant Permission[this, Math.sum] == 1.0 &&
        Heap[this, Math.sum] == oldHeap[this, Math.sum] +
        Heap[this, Math.i];
    invariant (forall<x> r:Ref, f : Field x :: (!(r==this && f==Math.i)) &&
        (!(r==this && f==Math.sum)) ==> Heap[r, f] == oldHeap[r, f]);
{
    assert Permission[this, Math.i] > 0.0;
    assert Permission[this, Math.sum] == 1.0;
    assert Isint32(Heap[this, Math.sum]);
    Heap[this, Math.sum] := Heap[this, Math.sum] + 1;
    assert Isint32(Heap[this, Math.sum]);
    Heap[this, Math.i] := Heap[this, Math.i] + 1;
    assert Isint32(Heap[this, Math.i]);
}
```

4.1.7 Threads

A Thread is a block of code declared inside a class. It can consist of statements and method implementations. It may be annotated with a **claim** specification to declare its initial permission on locations of the object.

The thread declaration is translated to a procedure in Boogie. The procedure has input parameter **this** denoting the object that the thread is running in. To track the permissions of a thread, local variables *Permission* and *ArrayPermission* are declared in the beginning of the generated procedure with zero initial permission. Also, *old* variables are declared which will be used to store the prior states in translation of statements. In the case that the thread was annotated with a **claim** specification, the permission variables receive the permissions on the specified locations. Also, we

assume the locations in **claim** specification, are equal to their declared initial values.

Finally, the code inside the thread is translated.

$Tr[(\textbf{thread}\ B\ \textbf{claim}\ init\_Permission\ block\ \textbf{thread})]_{H,sH,P} =$
  procedure $C.B(this : Ref)$
  modifies $H,\ ArrayH$;
  requires $dtype(this) <: C$;
  {
    var $Permission : PermissionType$ **where**
      (**forall** $\langle x \rangle\ r : Ref,\ f : Field\ x :: Permission[r, f] == 0.0$);
    var $ArrayPermission : ArrayPermissionType$ **where**
      (**forall** $\langle x \rangle\ r : ArrayRef\ x,\ f : \textbf{int} ::$
      $ArrayPermission[r, f] == 0.0$);
    var $oldH,\ H\_tmp : HeapType$;
    var $oldArrayH,\ ArrayH\_tmp : ArrayHeapType$;
    var $oldPermission : PermissionType$;
    var $oldArrayPermission : ArrayPermissionType$;

    $Tr[\textbf{claim}\ init\_Permission]_{H,oldH,Permission}$;
    $Tr^*[block]_{H,oldH,Permission}$;
  }

$Tr[\textbf{claim}\ init\_Permission]_{H,oldH,P} =$
  foreach $PermissionMap\ "PM"$ **in** $init\_Permission{\cdot}AddPermission(PM)_{H,oldH,P}$;
  $oldH := H$;
  $oldArrayH := ArrayH$;
  havoc H;
  havoc $ArrayH$;
  foreach $location\ E.f$ **in** $init\_Permission{\cdot}$
    assume $H[Tr[E]_{H,oldH,P},\ C.f] == Tr[Init_{E.f}]_{H,oldH,P}$;
  foreach $location\ E[f]$ **in** $init\_Permission{\cdot}$
    assume $ArrayH[Tr[E]_{H,oldH,P},\ Tr[f]_{H,oldH,P}] == Tr[Init_{E[f]}]_{H,oldH,P}$;
  assume (**forall** $\langle x \rangle\ r : Ref,\ f : Field\ x ::$
      $!(r.f\ \textbf{in}\ ModifiedVariable(init\_Permission)) ==>$
      $H[r, f] == oldH[r, f]$);
  assume (**forall** $\langle x \rangle\ r : ArrayRef\ x,\ f : int ::$
      $!(r[f]\ \textbf{in}\ ModifiedVariable(init\_Permission)) ==>$
      $ArrayH[r, f] == oldArrayH[r, f]$);

For instance, the translation of thread t1 of the Counter class in listing 4.0 is shown

below:

```
procedure Counter.t1(this : Ref)
modifies Heap, ArrayHeap;
requires dtype(this) <: Counter;
{
    var Permission : PermissionType where
        (forall <x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
    var ArrayPermission : ArrayPermissionType where
        (forall <x> r:ArrayRef x, f : int :: ArrayPermission [r, f] == 0.0);
    var oldHeap, Heap_tmp : HeapType;
    var oldArrayHeap, ArrayHeap_tmp : ArrayHeapType;
    var oldPermission : PermissionType;
    var oldArrayPermission : ArrayPermissionType;
    //claim
    Permission[this, Counter.count] := Permission[this, Counter.count] +
        1.0;
    oldHeap := Heap;
    havoc Heap;
    assume Heap[this, Counter.count] == 0; //intial value
    assume (forall<x> r:Ref, f : Field x :: !(r==this && f==Counter.count)
        ==> Heap[r, f] == oldHeap[r, f]);
        /* translation of body */
}
```

As shown, the thread is translated as procedure Counter.t1. First, two permission variables for fields and elements of arrays are declared. Next, the claim is translated which is full permission on count. Then the claimed location is assumed to have its initial value. Finally, the body of the thread is translated.

4.1.8 Methods

In HARPO/L, a method consists of a method declaration and method implementation. A method declaration is specified by contracts containing permission specifications and assertions. The permission specifications are declared with **takes** and

gives clauses. Furthermore, the assertions are declared with **pre** and **post** clauses which specify conditions on the variables. The syntax of a method declaration with specifications is shown below:

$MethodDeclararion \equiv$
proc $Name(Parameters)$
takes $in\_Permission$
pre $preconsition$
post $postcondition$
gives $out\_permission$

The method implementation is declared inside a thread and consists of a sequence of statements. Translation of a method implementation generates a block of code inside of a thread. To translate a method implementation, both specifications and implementation must be considered which is shown below.

$Tr[(\textbf{accept}\ M\ (Parameters)\ body\ \textbf{accept})]_{H,sH,P} =$
goto M;
$M:$
foreach $variable$ **in** $ins[Parameters]\cdot$
$\{HavocNewLoc(variable)_{H,sH,P};$
$AddPermission(variable@0.5)_{H,sH,P};\}$
foreach $variable$ **in** $outs[Parameters]\cdot$
$\{HavocNewLoc(variable)_{H,sH,P};$
$AddPermission(variable)_{H,sH,P};\}$
foreach $PermissionMap$ "PM" **in** $in\_Permission\cdot$
$\{HavocNewLoc(PM\langle variable\rangle)_{H,sH,P};$
$AddPermission(PM)_{H,sH,P};\}$
assert $Df[precondition]_{H,oldH,P}$;
$Assuming(precondition)_{H,oldH,P}$;
$preH := H$;
$preArrayH := ArrayH$;
$Tr^*[Body]_{H,-,P}$;
assert $Df[postcondition]_{preH,H,P}$;
assert $Tr[postcondition]_{preH,H,P}$;
foreach $variable$ **in** $outs\cdot$
$RemovePermission(variable)_{H,preH,P}$;
foreach $variable$ **in** $ins\cdot$

$RemovePermission(variable@0.5)_{H,preH,P}$;
foreach $PermissionMap$ "PM" **in** $out\_permission$·
$RemovePermission(PM)_{H,preH,P}$;
goto $Done$;
$Done$:

$Assuming(Cond)_{H,sH,P}$ =
$oldH := H$;
$oldArrayH := ArrayH$;
havoc H;
havoc $ArrayH$;
assume $Tr[Cond]_{H,oldH,P}$;
assume (**forall** $\langle x \rangle$ $r : Ref,\ f : Field\ x :: !(r.f$ **in** $ModifiedVariable(Cond))$
$==> H[r, f] == oldH[r, f]$);
assume (**forall** $\langle x \rangle$ $r : ArrayRef\ x,\ f : int :: !(r[f]$ **in** $ModifiedVariable(Cond))$
$==> ArrayH[r, f] == oldArrayH[r, f]$);

First, the thread recieves read permission on input parameters and write permission on output parameters. These locations are havocked to arbitrary values. In the translation, two functions *ins*[*Parameters*] and *outs*[*Parameters*] are used to extract the input and output parameters respectively.

$ins[Parameters] = \{p \mid$ **in** $p\ Prameters\}$
$outs[Parameters] = \{p \mid$ **out** $p\ Prameters\}$

Then, the permission specifications for pre-state are translated. The **takes** specification represents the permissions which will be transferred to the method from the caller at the beginning of the method. To translate **takes**, for each permission map specified in a **takes** clause, the permission variables of the thread are increased by those permissions. Also, all taken locations that thread had no permission on must be havocked too.

Second, the precondition is translated. It is asserted that the precondition is well-defined. Then the precondition is assumed which is shown by *Assuming* operation.

It havocs the heaps and then assumes the precondition. Also, it assumes that the locations not specified in the precondition, remain unchanged.

Third, the body of the method, which is a sequence of statements, is translated. Translation stores the value of the heaps prior to translation of body in *preH* and *preArrayH*, which will be used in two state postconditions. Then it translates the body of the method.

Fourth, the postconditions are translated. In fact, the implementation of the method must establish the postconditions. The translation asserts that the postconditions are well-defined and true.

Finally, the **gives** clause is translated. The **gives** specification represents the permissions which will be transferred from the method to the caller. To translate **gives**, for each permission map specified in **gives** clause, the permission variable of the thread is decreased by those permissions. Based on the fact that method parameters are defined only in the scope of the method, the permissions on parameters are removed in the end of method too.

Furthermore, the start of translation is labeled by the name of the method and there is a **goto** to the first of method's implementation in the beginning of translation. Also, the end of the accept block is labeled by *Done* and at the end of implementation of method, there is a **goto** to the end of the accept block. The reason for using labels is the fact that HARPO/L allows multiple methods to be implemented by one **accept** statement. To translate choice in an **accept** statement, the list of **goto**s at the beginning must include multiple method names. For instance, if there is a choice in an **accept** statement with methods $m0$ and $m1$, the **goto** list must include both of them: **goto** $m0$,$m1$;. We use the same translation methodology for the second

method.

For instance, translation of method increment of listing 4.0 is as follows:

```
goto increment;
increment:
//add permission
if (Permission[this, Counter.n] == 0.0)
      {havoc Heap_tmp;
       Heap[this, Counter.n] := Heap_tmp[this, Counter.n];}
Permission[this, Counter.n] := Permission[this, Counter.n] + 0.5;
if (Permission[this, Counter.count] == 0.0)
      {havoc Heap_tmp;
      Heap[this, Counter.count] := Heap_tmp[this, Counter.count];}
Permission[this, Counter.count] := Permission[this, Counter.count] + 1.0;
//precondition
oldHeap := Heap;
havoc Heap;
assume Heap[this, Counter.count] >= 0 && Heap[this, Counter.n] > 0;
assume Isint32(Heap[this, Counter.count] + Heap[this, Counter.n]);
assume (forall<x> r:Ref, f : Field x :: !((r==this && f == Counter.count) &&
   (r==this && f == Counter.n)) ==> Heap[r, f] == oldHeap[r, f]);
//body
preHeap := Heap;
preArrayHeap := ArrayHeap;
assert Permission[this, Counter.count] == 1.0;
assert Permission[this, Counter.n] > 0.0;
assert Isint32(Heap[this, Counter.count]);
assert Isint32(Heap[this, Counter.n]);
Heap[this, Counter.count] := Heap[this, Counter.count] +
      Heap[this, Counter.n];
assert Isint32(Heap[this, Counter.count]);
//postcondition
assert Heap[this, Counter.count] == preHeap[this, Counter.count] +
      Heap[this, Counter.n];
assert Isint32(Heap[this, Counter.count]);
//remove permission
assert Permission[this, Counter.count] == 1.0;
Permission[this, Counter.count] := Permission[this, Counter.count] - 1.0;
assert Permission[this, Counter.n] >= 0.5;
Permission[this, Counter.n] := Permission[this, Counter.n] - 0.5;
goto done_increment;
done_increment:
```

As shown, the start of the method is labeled by the name of the method. First, the thread receives read permission on the input parameter n and full permission on field count. These locations are havoced too. Then it assumes the precondition to be valid. To achieve this, it havocs the heap and assumes locations to have the values specified in the precondition. Also, it must be assumed that variables not mentioned by the precondition keep their values.

Next, the heaps are stored in *preH* and *preArrayH* and then the body is translated. It is checked that the thread has enough permission on variables count and n. Then it updates count with the new value.

Finally, the translation asserts the postcondition, which claims count has increased by n. Then the thread loses its permissions on count and n. In the end, there is a **goto** to done_increment which labels the end of the method's implementation.

4.1.9 Method Call

Translation of a call to method $E.M$ with input parameters and output parameters is shown below:

$Tr(E.M(Arguments))_{H,sH,P} =$
$preH := H;$
$preArrayH := ArrayH;$
assert $Df[E]_{H,preH,P};$
assert $Df^*[Arguments]_{H,preH,P};$
var $that : Ref;$
$that := E;$
foreach $param_i$ **in** $parameters \cdot$ **declare** $param_i;$
foreach $param_i$ **in** $ins \cdot param_i := Tr[arg_i]_{H,preH,P};$
assert $Df[precondition[\mathbf{this}/that]]_{H,-,P};$
assert $Tr[precondition[\mathbf{this}/that]]_{H,-,P};$
foreach $PermissionMap\ "PM"$ **in** $in\_Permission \cdot$
$RemovePermission(PM[\mathbf{this}/that])_{H,-,P};$

foreach $PermissionMap$ "PM" **in** $out\_Permission\cdot$
$\{HavocNewLoc(PM[\textbf{this}/that]\langle variable\rangle)_{H,-,P};$
$AddPermission(PM[\textbf{this}/that])_{H,-,P};\}$
assert $Df[postcondition[\textbf{this}/that]]_{preH,H,P};$
$Assuming(postcondition[\textbf{this}/that])_{preH,H,P};$
foreach $param_i$ **in** $outs$
$\{CanWrite(Tr[arg_i])_{H,preH,P};$
$Tr[arg_i]_{H,preH,P} := param_i;\}$

The translation first stores the value of the heaps in *preH* and *preArrayH* before a method call. Then it checks that all involving expressions are well-defined.

Establishing specifications of pre-state of a call is the responsibility of the caller. As a result, the precondition is checked to be defined and valid. Moreover a method loses permission on locations specified by the **takes** keyword. The specifications of a method are written from the method's point of view. As a result, before translating the specification, there is a replacement step. For the reference E and parameters of the method, new variables are declared. Prior to translating the precondition, the input arguments are copied to the associated variables. Then the specifications are translated based on these variables.

After a method call, the callee receives permissions for locations annotated by **gives**. These locations must be havoced if the thread had no permission on them. Then, it asserts the postcondition is well-defiined. Next, it assumes the postcondition. To achieve this, it havocs the heap and assumes the postcondition. Finally, the output variables are copied to output arguments.

For example, the translation of method call increment in listing 4.0 is as follows:

```
preHeap := Heap;
preArrayHeap := ArrayHeap;
that := this;
//copy arguments
```

```
assert Isint32(Heap[this, Counter.i]);
n := Heap[this, Counter.i];
assert Isint32(n);
//precondition
assert Heap[that, Counter.count] >= 0 && n > 0;
assert Isint32(Heap[that, Counter.count] + n);
//remove permission
assert Permission[this, Counter.count] == 1.0;
Permission[this, Counter.count] := Permission[this, Counter.count] - 1.0;
//add permission
if (Permission[this, Counter.count] == 0.0)
    {havoc Heap_tmp;
    Heap[this, Counter.count] := Heap_tmp[this, Counter.count];}
Permission[this, Counter.count] := Permission[this, Counter.count] + 1.0;
//postcondition
oldHeap := Heap;
havoc Heap;
assume (Heap[that, Counter.count] == preHeap[that, Counter.count] + n);
assume (forall<x> t:Ref, f : Field x :: (!(t==that && f==Counter.count)
    ==> Heap[t,f] == oldHeap[t,f]));
assert Heap[this, Counter.count] == 1;
```

The translation stores the value of the heaps in the beginning of translating the method call. A new variable, "that" is declared to store the object reference of the method. Also, variable n is declared which stores the value of argument i. Next, the precondition is asserted; this checks the value of count and n. Then, the thread loses the permission required by the method, which is full permission on count.

After the method call, the thread receives the returning permission, which is full permission on count. Also, this location is havoced. Next, the postcondition is translated. It assumes the field count has been incremented by n and other fields have remained unchanged. In this example, the value of count was zero before the method call and it is asserted it is one after method call.

4.1.10 Constructor

The constructor of a class is translated to a procedure in Boogie. The input parameter of the procedure is this denoting the class. The constructor is allowed to modify all fields of the class. Translation of a constructor is illustrated in below:

$Tr[$**constructor** $(Parameters)]_{H,sH,P} =$
procedure $C.C$ $(Ref\ this)$
modifies $H,\ ArrayH$;
modifies $LockPermission,\ ArrayLockPermission$;
free requires $this <: C$;
{
var $Permission : PermissionType$ **where**
(**forall** $\langle x \rangle\ r : Ref,\ f : Field\ x :: Permission[r, f] == 0.0$);
var $ArrayPermission : ArrayPermissionType$ **where**
(**forall** $\langle x \rangle\ r : ArrayRef\ x,\ f :$ **int** ::
$ArrayPermission[r, f] == 0.0$);
var $oldH : HeapType$;
var $oldArrayH : ArrayHeapType$;
$oldH := H$;
$oldArrayH := ArrayH$;
havoc H;
havoc $ArrayH$;
//add permission on in parameters
foreach $PermissionMap$ "PM" **in** $ins[Parameters]\cdot$
$AddPermission(PM@0.5)_{H,-,Permission}$;
//check claims
foreach $field$ "f" $defined$ **in** C **do**
assert $sum\ of\ claimed\ permission\ on$ "f" $<= 1.0$;
//initializing fields
foreach $field$ "f" $defined$ **in** C **do**
$\{Tr[f]_{H,-,Permission} := Tr[InitExp_f]_{H,-,Permission};\}$
// check class claim
foreach $PermissioMap$ "PM" **in** $the\ claim\ of\ the\ class$ "**claim** PMs"$\cdot$
$AddPemission(PM)_{H,oldH,LockPermission}$;
//invariant
assert $Df[J]_{H,oldH,LockPermission}$;
assert $Tr[J]_{H,oldH,LockPermission}$;
}

$Tr[\textbf{acc}\ PMs]_{H,sH,P} =$
    foreach $PermissioMap$ "PM" **in** $PMs \cdot CanAccess(PM)_{H,sH,P}$;

First, the translation havocs the heaps which gives arbitrary values to the heap locations. Second, the translation adds permission on the input parameters of constructor. Third, the translation checks that the sum of claims on each field is not more than one. Fourth, it initializes the fields of the class with their declared initial values.

Fifth, it translates the **claim** of the class which increases the permission of the lock based on permission maps in the **claim** specification. Finally, it asserts that the invariant is well-defined and is valid. An invariant has two types of specification. One is permission specification, annotated with **acc** keyword. The second type is pure assertions, declaring conditions on variables. The translation of **acc** is shown in translation too, which checks having the specified permission.

For instance, the translation of the constructor of Counter class in listing 4.0 is shown below:

```
procedure Counter.Counter(this:Ref)
requires dtype(this) <: Counter;
modifies Heap;
modifies LockPermission;
{
    var Permission : PermissionType where
        (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
    var ArrayPermission : ArrayPermissionType where
        (forall<x> r:ArrayRef x, f : int :: ArrayPermission [r, f] == 0.0);
    var oldHeap : HeapType;
    var oldArrayHeap: ArrayHeapType;
    oldHeap := Heap;
    oldArrayHeap := ArrayHeap;
    havoc Heap;
    havoc ArrayHeap;
```

```
    //check claims
    Permission[this, Counter.count] := Permission[this, Counter.count] +
        1.0;
    assert (forall<x> r:Ref, f : Field x :: Permission[r, f] <= 1.0);
    //initialize
    Heap[this, Counter.count] := 0;
    assert Isint32(Heap[this, Counter.count]);
}
```

In this example, first it havocs the heaps in entrance to constructor. Next, it updates the Permission variable based on declared claims and checks that the sum of claims on each location is not greater than one. In this example, there is one **claim** specification and its claimed permission is not greater than one. Next, it initializes the field count with its initial value.

4.1.11 Locks

Translation of locks is done based on the invariant of the class. Assuming J is the invariant of a class, the translation of a lock is shown in below:

$Tr[($**with** l
takes $in\_Permission$
gives $out\_Permission$
$stmts)]_{H,oldH,P} =$
var $that : Ref;$
$that := Tr[l]_{H,oldH,LockPermission};$
$AssumingPermission(J_{acc}[this/that])_{H,oldH,LockPermission};$
assert $Df[J_{cond}[this/that]]_{H,oldH,LockPermission};$
$Assuming(J_{cond}[this/that])_{H,oldH,LockPermission};$
foreach $PermissionMap\ "PM"$ **in** $in\_Permission\cdot$
$RemovePermission(PM)_{H,oldH,LockPermission};$
foreach $PermissionMap\ "PM"$ **in** $in\_Permission\cdot$
$AddPermission(PM)_{H,oldH,P};$
$//body$
$Tr^*[stmts]_{H,oldH,P+LockPermission};$
foreach $PermissionMap\ "PM"$ **in** $out\_Permission\cdot$

$RemovePermission(PM)_{H,oldH,P}$;
foreach $PermissionMap$ "PM" **in** $out\_Permission$·
$AddPermission(PM)_{H,oldH,LockPermission}$;
assert $Df[J[this/that]]_{H,oldH,LockPermission}$;
assert $Tr[J[this/that]]_{H,oldH,LockPermission}$;

$AssumingPermission(PermCond)_{H,oldH,LockP} =$
havoc $LockP$;
havoc $ArrayLockP$;
foreach $PermissionMap$ "PM" **in** $PermCond$·
$\{HavocNewLoc(PM\langle variable\rangle)_{H,oldH,ZeroP}$;
assume $CanAccess(PM)_{H,oldH,LockP}$;$\}$
assume (**forall** $\langle x\rangle$ $r : Ref,\ f : Field\ x$ ::!$(r.f$ **in** $ModifiedVariable(PermCond))$
$==>$ $LockP[r, f] == 0.0)$;
assume (**forall** $\langle x\rangle$ $r : ArrayRef\ x,\ f : int$ ::!$(r[f]$ **in** $ModifiedVariable(PermCond))$
$==>$ $ArrayLockP[r, f] == 0.0)$;

First, the invariant is assumed to be true on entrance to the lock. A new variable *that* is declared to store the reference to the lock. It will replace *this* in translation of the invariant. An invariant has two kinds of specifications, permission specifications, declared with the **acc** keyword, and condition specifications. Translation of the permission part is shown by the *AssumingPermission* operation which assumes the specified permissions. These locations must be havoced to arbitrary values too. We also assume that the permission of locations not specified in **acc** specifications are zero.

After assuming the permission part of the invariant, we can translate the condition part of invariant. It is asserted that the condition part is defined. Then, the condition part is assumed.

A lock may be annotated with a **takes** specification. This leads to transferring specified permissions from the lock to the thread. As shown in the translation, the

lock loses permission and the thread receives those permissions. Next, the statements inside the lock are translated. Within this translation, the permissions of the thread and lock must be summed for checks of readability and writability. Also, a lock can be annotated with a **gives** specification, which leads to transferring specified permissions from the thread to the lock. As shown in the translation, the thread loses the specified permissions and the lock receives them.

Finally, the translation must check that the invariant holds on release of the lock. It asserts that the lock permission variables hold the specified permissions. Also, it asserts the condition assertions of the invariant are defined and true.

The translation of lock is shown in listing 3.6. First, the translation havocs the locations on which the thread has no permission. Then, it assumes the invariant is defined and true. Next, it translates the assignment by considering the permissions of both thread and lock. Finally, it asserts the invariant.

```
that := this;
//assume invariant
//acc count@0.5
havoc LockPermission;
havoc Heap_tmp;
Heap[that, Counter.count] := Heap_tmp[that, Counter.count];
assume LockPermission[that, Counter.count] == 0.5;
assume (forall<x> r:Ref, f: Field x :: !(r==that && f==Counter.count) ==>
    LockPermission[r, f] == 0.0);
//0 <= count
oldHeap := Heap;
havoc Heap;
assume Heap[that, Counter.count] >= 0.0;
assume (forall<x> r:Ref, f : Field x :: !(r==that && f == Counter.count)
    ==> Heap[r, f] == oldHeap[r, f]);
//lock body
assert ((Permission[this, Counter.count] +
    LockPermission[this, Counter.count]) == 1.0);
Heap[this, Counter.count] := Heap[this, Counter.count] + 1.0;
```

//assert inv
assert LockPermission[that, Counter.count] == 0.5;
assert Heap[that, Counter.count] >= 0.0;

4.1.12 Parallelism

Parallel blocks may be specified with a **claim** specification, denoting their required permissions. At first, the main thread loses permission according to claimed permissions by parallel blocks. Then the **co** blocks are translated respectively.

$Tr[(\textbf{co claim } claim1\ stmts1 \mid\mid \textbf{claim } claim2\ stmts2\ \textbf{co})]_{H,oldH,P} =$
$foreach\ PermissionMap\ "PM"\ in\ claim1 \cdot RemovePermission(PM)_{H,oldH,P};$
$foreach\ PermissionMap\ "PM"\ in\ claim2 \cdot RemovePermission(PM)_{H,oldH,P};$
var $Permission1 : PermissionType;$
havoc $Permission1;$
assume (**forall** $\langle x \rangle\ r : ref,\ f : Field\ x :: Permission1[r, f] == 0.0);$
foreach $PermissionMap\ "PM"$ **in** $claim1 \cdot$
    $AddPermission(PM)_{H,oldH,Permission1};$
$Tr^*[stmts1]_{H,oldH,Permission1};$
var $Permission2 : PermissionType;$
havoc $Permission2;$
foreach $PermissionMap\ "PM"$ **in** $claim2 \cdot$
    $AddPermission(PM)_{H,oldH,Permission2};$
$Tr^*[stmts2]_{H,oldH,Permission2};$
$oldP := P;$
$oldArrayP := ArrayP;$
foreach $location\ "lc"$ **in** $Permission1 \cdot$
    $AddPermission(lc@Permission1(lc))_{H,oldH,P};$
foreach $location\ "lc"$ **in** $Permission2 \cdot$
    $AddPermission(lc@Permission2(lc))_{H,oldH,P};$

To translate the first **co** block, the new variable, *Permission*1, is declared to track permission. If the **co** block is prefixed with a **claim**, the permission variable is updated according to the claimed permissions. Then the statements of the child thread are translated using only the *Permission*1 variable to check read and write

access. This procedure is repeated for the remaining **co** blocks. At the end, the permission variable of the main thread receives the remaining permissions of the **co** blocks.

The translation of the **co** statement in listing 3.7 is shown below. In the translation of **co** statements, the parent thread loses the claimed permissions on variables a, sum1 and sum2. Then the co1 block is translated. It receives read permission on a and full permission on sum1. Then the co2 block is translated with the same methodology. In the end, the main thread receives the permissions of both threads.

```
//parent thread t0
assert Permission[this, Math.a] >= 0.5;
Permission[this, Math.a] := Permission[this, Math.a] - 0.5;
assert Permission[this, Math.sum1] == 1.0;
Permission[this, Math.sum1] := Permission[this, Math.sum1] - 1.0;
assert Permission[this, Math.a] >= 0.25;
Permission[this, Math.a] := Permission[this, Math.a] - 0.25;
assert Permission[this, Math.sum2] == 1.0;
Permission[this, Math.sum2] := Permission[this, Math.sum2] - 1.0;
Permission1[this, Math.a] := Permission1[this, Math.a] + 0.5; //co1
Permission1[this, Math.sum1] := Permission1[this, Math.sum1] + 1.0;
assert Permission1[this, Math.a] > 0.0 ;
assert Permission1[this, Math.sum1] == 1.0;
assert Isint32(Heap[this, Math.a]);
Heap[this, Math.sum1] := Heap[this, Math.a] + 1;
assert Isint32(Heap[this, Math.sum1]);
Permission2[this, Math.a] := Permission2[this, Math.a] + 0.25; //co2
Permission2[this, Math.sum2] := Permission2[this, Math.sum2] + 1.0;
assert Permission2[this, Math.a] > 0.0;
assert Permission2[this, Math.sum2] == 1.0;
assert Isint32(Heap[this, Math.a]);
Heap[this, Math.sum2] := Heap[this, Math.a] + 1;
assert Isint32(Heap[this, Math.sum2]);
//Add permission to parent thread
oldPermission := Permission;
havoc Permission;
assume (forall<x> r:Ref, f: Field x :: (Permission[r, f] ==
    oldPermission[r, f] + Permission1[r, f] + Permission2[r, f]));
```

Chapter 5

Case Studies

This chapter presents some examples in HARPO/L with their verification. The first case shows verification of a program implementing a buffer. The second case presents a merging algorithm. The third one verifies three scenarios of permission transfer.

5.0 Buffer

In this example, a buffer with specifications is shown. The buffer is implemented by an array. Two operations are declared on buffer: *deposit* and *fetch*. To synchronize these operations, the variable *full* is declared. When the value of *full* is less than the buffer size, the *deposit* operation is allowed. When the value of *full* is greater than zero, the *fetch* operation is permitted. The *front* index shows the slot to be fetched and the *rear* index shows the first empty slot. The code in listing 5.0 shows the implementation.

```
(class Buffer

    proc deposit(in value : real)
```

```
    proc fetch(out value : real)

    obj buf : real[size] := (for i : size do 0 for)
    obj front : int32 := 0
    obj rear : int32 := 0
    obj full : int32 := 0
    const size : int32 := 10

    (thread (*t0*) claim front, rear, full, {i:{0,..size} . buf[i]}
        (while true
            invariant acc front, rear, full, {i:{0,..size} . buf[i]}
            invariant (0 _< front /\ front < size) /\
                      (0 _< rear /\ rear < size) /\
                      (0 _< full /\ full < size)
            invariant ((front + full) mod size = rear)
            (accept deposit(in value : real) when (full < size)
                    buf[rear] := value
                    rear := (rear + 1) % size
                    full := full + 1
            |
            fetch(out ovalue : real) when (0 < full)
                    ovalue := buf[front]
                    front := (front + 1) % size
                    full := full - 1
            accept)
        while)
    thread)
class)
```

Listing 5.0: Buffer class

In this code, the thread t0 has full permission on the indices and the array. This allows modification of them by the methods. Also, the loop is specified with invariants. The first invariant asserts full permission on front, rear, full and the array. The second invariant declares the range of front, rear and full. The last invariant declares the relation between front, rear and full. The translation of this code is shown in listing 5.1.

```
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);

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;

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] == 1.0);
      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 values 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] == 0.0);
      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 (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.ovalue)
                    ==> 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]);
      {
        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.rear]);
            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.ovalue] := ArrayHeap[Heap[this, Buffer.buf], Heap[this, ↩
↪Buffer.front]];
          assert Permission[this, Buffer.front] == 1.0;
          assert 0 <= Heap[this, Buffer.front] && Heap[this, Buffer.front] < ↩
↪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 5.1: Translation of buffer class in Boogie

The translation code is given as input to Boogie verifier. The Boogie verifier reports successful verification and gives the following message.

Boogie program verifier finished with 1 verified, 0 errors

In addition to verifying a correct program, we are interested in introducing errors into the program to see whether the verifier detects the errors. First, we introduce errors into the program and check the result of verification. Second, we introduce errors into the specification and then test the result.

- **Erroneous Code**

An array access must be in the range of array bounds. We modified "ovalue := buf[front]" to instead use an incorrect index: "ovalue := buf[front+1]". Accessing this location is not allowed and can not establish the definedness of array access and gives the following error:

Error BP5001: This assertion might not hold.

- **Erroneous Specification**

For the second test, the code is correct but we introduce errors into the specification. In this example, there are two types of specifications: claims and loop invariants.

Writing the loop invariants is more challenging, because the only way that the verifier will know that the properties of the loop is preserved is by the invariants. For instance, we change the invariant "((front + full) mod size = rear)" to "((front + full) mod size < rear)". The verifier gives an error on this line and the program is not verified.

5.1 Merge

In this example, a merge algorithm is implemented. As shown in listing 5.2, the method merges two sorted segments of array a, to one sorted segment. The first segment is from index left to right-1 and the second one is from index right to end. The precondition declares that these sublists are sorted and request write permission on the array from element left to end. The method combines these sublists to one sorted list from left to end in array b. The while statement is annotated with loop invariants. The invariants declare the range of used indices. Also, it is declared that b is sorted. Finally, the sorted array b is copied in array a. The method asserts that the array is sorted from left to end and then loses permission on these locations.

```
(class M

    (in length : int32, obj a, b : real[length])
    pre a != b
    pre 0 < length

    proc merge(in left : int32, in right : int32, in end : int32)
    pre 0 _< left /\ left < right /\ right _< end /\ end < Length(a)
    takes { i : {left,..,end} . a(i)}
    takes { i : {left,..,end} . b(i)}
    pre (forall i, j : left _< i /\ i < j /\ j < right . a[i] _< a[j])
    pre (forall i, j : right _< i /\ i < j /\ j _< end . a[i] _< a[j])
    post (forall i, j : left _< i /\ i < j /\ j _< end . a[i] _< a[j])
```

```
    gives { i : {left,..,end} . a(i)}
    gives { i : {left,..,end} . b(i)}

    (thread (*t0*)
        (accept merge(in left : int32, in right : int32, in end : int32)
            obj ileft : int32 := left
            obj iright : int32 := right
            obj k : int32 := left
            (while (left _< k /\ k _< end)
                invariant acc left@0.5, right@0.5, end@0.5, k, ileft, iright,
                    {i:{left,..,end} . a[i]}, {i:{left,..,end} . b[i]}
                invariant (left _< k /\ k _< end+1)
                invariant (left _< ileft /\ ileft _< right)
                invariant (right _< iright /\ iright _< end+1)
                invariant (forall i, j : (left _< i /\ i < k /\ ileft _< j /\ j <-
->< right). b[i] _< a[j])
                invariant (forall i, j : (left _< i /\ i < k /\ iright _< j /\ j <-
->_< end) . b[i] _< a[j])
                invariant (forall i, j : (left _< i /\ i _< j /\ j < k) . b[i] _<-
->< b[j]);
                (if (ileft < right /\ iright _< end /\ k _< end)
                    (if (a[ileft] _< a[iright])
                        b[k] := a[ileft];
                        ileft := ileft + 1
                     (else
                        b[k] := a[iright]
                        iright := iright + 1
                     )
                     if)
                     k := k + 1
                  (else if (ileft = right /\ iright _< end /\ k _< end)
                    b[k] := a[iright]
                    iright := iright + 1
                    k := k + 1
                  )
                  (else if (iright = end+1 /\ ileft < right /\ k _< end)
                    b[k] := a[ileft]
                    ileft := ileft + 1
                    k := k + 1
                  )
                  if)
                while)
```

```
                k := left
                (while (left _< k /\ k _< end)
                    invariant acc left@0.5, end@0.5, k,
                          {i:{left,..,end} . a[i]}, {i:{left,..,end} . b[i]}
                    invariant (left _< k /\ k _< end+1)
                    invariant (forall i : (left _< i /\ i < k) ==> a[i] = b[i]);
                    a[k] := b[k]
                    k := k + 1
                while)
            accept)
    thread)
class)
```

Listing 5.2: Merge code

The translation of the merge method is provided in listing 5.3. It verifies that array a is sorted from left to end.

```
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);
axiom 0 < Length(M.a);
axiom 0 < Length(M.b);
axiom Length(M.a) == Length(M.b);

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

```
const unique M : ClassName;
const unique M.left : Field int;
const unique M.right : Field int;
const unique M.end : Field int;
const unique M.size : Field int;
const unique M.ileft : Field int;
const unique M.iright : Field int;
const unique M.k : Field int;
const unique M.a: Field (ArrayRef int);
const unique M.b: Field (ArrayRef int);

procedure t0(this:Ref)
  modifies Heap, ArrayHeap;
  requires Heap[this, M.a] != Heap[this, M.b];
{
  var Permission, oldPermission, prePermission : PermissionType;
  var ArrayPermission, oldArrayPermission, preArrayPermission : ↩
↪ArrayPermissionType;
  var oldHeap, preHeap, tmp_Heap : HeapType;
  var oldArrayHeap, preArrayHeap, tmp_ArrayHeap : ArrayHeapType;

  havoc Permission;
  assume (forall<x> o:Ref, f : Field x :: Permission[o, f] == 0.0);
  havoc ArrayPermission;
  assume (forall<x> o:ArrayRef x, f : int :: ArrayPermission[o, f] == 0.0);
  prePermission := Permission;
  preArrayPermission := ArrayPermission;
  while(true)
    invariant Heap[this, M.a] != Heap[this, M.b];
    invariant (forall<x> o:Ref, f : Field x :: Permission[o, f] == prePermission[o, ↩
↪f]);
    invariant (forall<x> o:ArrayRef x, f : int :: ArrayPermission[o, f] == ↩
↪preArrayPermission[o, f]);
  {
      //merge
    goto Merge;
    Merge:
      //permission on parameters
      havoc tmp_Heap;
      Heap[this, M.left] := tmp_Heap[this, M.left];
      Permission[this, M.left] := Permission[this, M.left] + 0.5;
      havoc tmp_Heap;
```

```
        Heap[this, M.right] := tmp_Heap[this, M.right];
        Permission[this, M.right] := Permission[this, M.right] + 0.5;
        havoc tmp_Heap;
        Heap[this, M.end] := tmp_Heap[this, M.end];
        Permission[this, M.end] := Permission[this, M.end] + 0.5;
        //pre 0 <= left && left < right && right <= M.end && M.end < ↩
↪a.Length
        oldHeap := Heap;
        havoc Heap;
        assume (0 <= Heap[this, M.left] && Heap[this, M.left] < Heap[this, M.right↩
↪] && Heap[this, M.right] <= Heap[this, M.end] &&
                Heap[this, M.end] < Length(M.a));
        assume (forall<x> o:Ref, f : Field x :: !((o == this && f == M.left) || (o ↩
↪== this && f == M.right) || (o == this && f == M.end)) ==>
            Heap[o, f]==oldHeap[o, f]);
        //takes { i : {left,..M.end} . a(i) }
        oldArrayPermission := ArrayPermission;
        havoc ArrayPermission;
        assume (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
            ArrayPermission[o,f] == oldArrayPermission[o,f] + 1.0);
        assume (forall<x> o:ArrayRef x, f : int :: !((o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end]))==>
            ArrayPermission[o,f] == oldArrayPermission[o,f]);
        //takes { i : {left,..M.end} . b(i) }
        oldArrayPermission := ArrayPermission;
        havoc ArrayPermission;
        assume (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
            ArrayPermission[o,f] == oldArrayPermission[o,f] + 1.0);
        assume (forall<x> o:ArrayRef x, f : int :: !((o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end]))==>
            ArrayPermission[o,f] == oldArrayPermission[o,f]);
        //sorted a(left,..right) and a(right,..M.end)
        assert 0.0 < Permission[this, M.left] && 0.0 < Permission[this, M.right] && ↩
↪0.0 < Permission[this, M.end];
        oldArrayHeap := ArrayHeap;
        havoc ArrayHeap;
        assume (forall i : int, j:int :: (Heap[this, M.left] <= i && i < j && j < ↩
↪Heap[this, M.right]) ==>
          ArrayHeap[Heap[this, M.a], i] <= ArrayHeap[Heap[this, M.a], j]);
        assume (forall i : int, j:int :: (Heap[this, M.right] <= i && i < j && j <=↩
```

```
↪ Heap[this, M.end]) ==>
        ArrayHeap[Heap[this, M.a], i] <= ArrayHeap[Heap[this, M.a], j]);
      assume (forall<x> i : int, t:ArrayRef x :: !(t==Heap[this, M.a] && (Heap[↩
↪this, M.left] <= i && i <= Heap[this, M.end])) ==>
        ArrayHeap[t, i] == oldArrayHeap[t, i]);
      //body
      preHeap := Heap;
      preArrayHeap := ArrayHeap;
      //local variables
      Permission[this, M.ileft] := Permission[this, M.ileft] + 1.0;
      assert Permission[this, M.left] > 0.0;
      Heap[this, M.ileft] := Heap[this, M.left];
      Permission[this, M.iright] := Permission[this, M.iright] + 1.0;
      assert Permission[this, M.right] > 0.0;
      Heap[this, M.iright] := Heap[this, M.right];
      Permission[this, M.k] := Permission[this, M.k] + 1.0;
      assert Permission[this, M.left] > 0.0;
      Heap[this, M.k] := Heap[this, M.left];
        oldPermission := Permission;
        oldArrayPermission := ArrayPermission;
      oldArrayHeap := ArrayHeap;
      oldHeap := Heap;

      while (Heap[this, M.left] <= Heap[this, M.k] && Heap[this, M.k] <= Heap[↩
↪this, M.end])
       invariant Permission[this, M.left] == 0.5 && Permission[this, M.right] == ↩
↪0.5 && Permission[this, M.end] == 0.5 &&
                 Permission[this, M.k] == 1.0 && Permission[this, M.ileft] == 1.0 ↩
↪&& Permission[this, M.iright] == 1.0 &&
                 (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
                      ArrayPermission[o,f] == 1.0) &&
                 (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
                      ArrayPermission[o,f] == 1.0);
       invariant (forall<x> r:Ref, f : Field x :: (!(r==this && f == M.left) && ↩
↪!(r==this && f == M.right) &&
                  !(r==this && f == M.end) && !(r==this && f == M.ileft) ↩
↪&& !(r==this && f == M.iright) && !(r==this && f == M.k) )
                  ==> Permission[r, f] == oldPermission[r, f]);
       invariant (forall<x> r:ArrayRef x, f : int :: !(((r == Heap[this, M.a])||(r ↩
↪== Heap[this, M.b])) && (Heap[this, M.left] <= f && f <= Heap[this, M.end]))↩
```

```
↪==>
                    ArrayPermission[r,f] == oldArrayPermission[r,f]);
                invariant (Heap[this, M.left] <= Heap[this, M.k] && Heap[this, M.k]↩
↪ <= Heap[this, M.end]+1);
                invariant (Heap[this, M.left] <= Heap[this, M.ileft] && Heap[this, ↩
↪M.ileft] <= Heap[this, M.right]);
                invariant (Heap[this, M.right] <= Heap[this, M.iright] && Heap[this,↩
↪ M.iright] <= Heap[this, M.end]+1);
                invariant (forall<x> o:Ref, f : Field x :: (!((o == this && f == ↩
↪M.ileft) || (o == this && f == M.iright)|| (o == this && f == M.k))) ==>
                  Heap[o, f]==oldHeap[o, f]);
                invariant (forall i : int, j : int :: (Heap[this, M.left] <= i && i < ↩
↪Heap[this, M.k] && Heap[this, M.ileft] <= j && j < Heap[this, M.right]) ==>
                  ArrayHeap[Heap[this, M.b],i] <= ArrayHeap[Heap[this, M.a],j]);
                invariant (forall i : int, j : int :: (Heap[this, M.left] <= i && i < ↩
↪Heap[this, M.k] && Heap[this, M.iright] <= j && j <= Heap[this, M.end]) ==>
                  ArrayHeap[Heap[this, M.b],i] <= ArrayHeap[Heap[this, M.a],j]);
                invariant (forall<x> i, j: int, t:ArrayRef x :: (t == Heap[this, M.b] ↩
↪&& (Heap[this, M.left] <= i && i < j && j < Heap[this, M.k])) ==>
                  ArrayHeap[Heap[this, M.b], i] <= ArrayHeap[Heap[this, M.b], j]);
                invariant (forall<x> i:int, t:ArrayRef x :: (t !=Heap[this, M.b] || !(↩
↪Heap[this, M.left] <= i && i < Heap[this, M.k]))==>
                  ArrayHeap[t,i] == oldArrayHeap[t,i]);
        {
          assert 0.0 < Permission[this, M.left] && 0.0 < Permission[this, M.right] ↩
↪&& 0.0 < Permission[this, M.end] &&
                  0.0 < Permission[this, M.ileft] && 0.0 < Permission[this, M.iright] ↩
↪&& 0.0 < Permission[this, M.k];
          if (Heap[this, M.ileft] < Heap[this, M.right] &&
                  Heap[this, M.iright] <= Heap[this, M.end] &&
                  Heap[this, M.k] <= Heap[this, M.end])
           {
              assert 0.0 < Permission[this, M.ileft] && 0.0 < Permission[this, ↩
↪M.iright] &&
                      0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.ileft]] ↩
↪&&
                      0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.iright]];
              if (ArrayHeap[Heap[this, M.a], Heap[this, M.ileft]] <= ArrayHeap[↩
↪Heap[this, M.a], Heap[this, M.iright]])
              {
                assert 0.0 < Permission[this, M.k] && 0.0 < Permission[this, ↩
↪M.ileft] &&
```

```
                    0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.ileft]] ↩
↪&&
                    ArrayPermission[Heap[this, M.b], Heap[this, M.k]] == 1.0 ↩
↪&&
                    0 <= Heap[this, M.k] && Heap[this, M.k] < Length(M.b);
                ArrayHeap[Heap[this, M.b], Heap[this, M.k]] := ArrayHeap[Heap[↩
↪this, M.a], Heap[this, M.ileft]];
                assert Permission[this, M.ileft] == 1.0;
                Heap[this, M.ileft] := Heap[this, M.ileft] + 1;
              }
              else
              {
                assert 0.0 < Permission[this, M.k] && 0.0 < Permission[this, ↩
↪M.iright] &&
                    0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.iright↩
↪]]&&
                    ArrayPermission[Heap[this, M.b], Heap[this, M.k]] == 1.0;
                assert 0 <= Heap[this, M.k] && Heap[this, M.k] < Length(M.b);
                ArrayHeap[Heap[this, M.b], Heap[this, M.k]] := ArrayHeap[Heap[↩
↪this, M.a], Heap[this, M.iright]];
                assert Permission[this, M.iright] == 1.0;
                Heap[this, M.iright] := Heap[this, M.iright] + 1;
              }
              assert Permission[this, M.k] == 1.0;
              Heap[this, M.k] := Heap[this, M.k] + 1;
            }
            else if (Heap[this, M.ileft] == Heap[this, M.right] &&
                  Heap[this, M.iright] <= Heap[this, M.end] &&
                        Heap[this, M.k] <= Heap[this, M.end])
            {
              assert 0.0 < Permission[this, M.k] && 0.0 < Permission[this, M.iright]↩
↪ &&
                    0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.iright]] &&
                    ArrayPermission[Heap[this, M.b], Heap[this, M.k]] == 1.0;
              assert 0 <= Heap[this, M.k] && Heap[this, M.k] < Length(M.b);
              ArrayHeap[Heap[this, M.b], Heap[this, M.k]] := ArrayHeap[Heap[this, ↩
↪M.a], Heap[this, M.iright]];
              assert Permission[this, M.iright] == 1.0;
              Heap[this, M.iright] := Heap[this, M.iright] + 1;
              assert Permission[this, M.k] == 1.0;
              Heap[this, M.k] := Heap[this, M.k] + 1;
            }
```

```
            else if ((Heap[this, M.iright] == Heap[this, M.end]+1) &&
                  Heap[this, M.ileft] < Heap[this, M.right] &&
                  Heap[this, M.k] <= Heap[this, M.end])
            {
              assert 0.0 < Permission[this, M.k] && 0.0 < Permission[this, M.ileft] ↩
↪&&
                     0.0 < ArrayPermission[Heap[this, M.a], Heap[this, M.ileft]] &&
                     ArrayPermission[Heap[this, M.b], Heap[this, M.k]] == 1.0;
              assert 0 <= Heap[this, M.k] && Heap[this, M.k] < Length(M.b);
              ArrayHeap[Heap[this, M.b], Heap[this, M.k]] := ArrayHeap[Heap[this, ↩
↪M.a], Heap[this, M.ileft]];
              assert Permission[this, M.ileft] == 1.0;
              Heap[this, M.ileft] := Heap[this, M.ileft] + 1;
              assert Permission[this, M.k] == 1.0;
              Heap[this, M.k] := Heap[this, M.k] + 1;
            }
        }

        assert Permission[this, M.k] == 1.0 && 0.0 < Permission[this, M.left];
        Heap[this, M.k] := Heap[this, M.left];
        assert ArrayPermission[Heap[this, M.a], Heap[this, M.k]] == 1.0;
        oldArrayHeap := ArrayHeap;
        oldHeap := Heap;
            oldPermission := Permission;
            oldArrayPermission := ArrayPermission;
        assert 0.0 < Permission[this, M.left] && 0.0 < Permission[this, M.k] && 0.0 ↩
↪< Permission[this, M.end];
        while ((Heap[this, M.left] <= Heap[this, M.k] && Heap[this, M.k] <= Heap[↩
↪this, M.end]))
        invariant Permission[this, M.left] == 0.5 && Permission[this, M.end] == 0↩
↪.5 && Permission[this, M.k] == 1.0 &&
                        (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
                              ArrayPermission[o,f] == 1.0) &&
                        (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
                              ArrayPermission[o,f] == 1.0);
        invariant (forall<x> r:Ref, f : Field x :: (!(r==this && f == M.left) && !(↩
↪r==this && f == M.k) && !(r==this && f == M.end))
                         ==> Permission[r, f] == oldPermission[r, f]);
        invariant (forall<x> r:ArrayRef x, f : int :: !(((r == Heap[this, M.a])||(r ↩
↪== Heap[this, M.b])) && (Heap[this, M.left] <= f && f <= Heap[this, M.end]))↩
```

```
↪==>
                    ArrayPermission[r,f] == oldArrayPermission[r,f]);
        invariant (Heap[this, M.left] <= Heap[this, M.k] && Heap[this, M.k] ↩
↪<= Heap[this, M.end]+1);
        invariant (forall<x> o:Ref, f : Field x :: !(o == this && f == M.k) ↩
↪==> Heap[o, f] == oldHeap[o, f]);
        invariant (forall i : int :: (Heap[this, M.left] <= i && i < Heap[this, ↩
↪M.k]) ==> ArrayHeap[Heap[this, M.a], i] == ArrayHeap[Heap[this, M.b], i]);
        invariant (forall<x> i : int , t : ArrayRef x :: !(t ==Heap[this, M.a] ↩
↪&& (Heap[this, M.left] <= i && i < Heap[this, M.k])) ==>
          ArrayHeap[t,i] == oldArrayHeap[t,i]);
    {
        assert 0.0 < Permission[this, M.k] &&
            ArrayPermission[Heap[this, M.a], Heap[this, M.k]] == 1.0 &&
            ArrayPermission[Heap[this, M.b], Heap[this, M.k]] == 1.0 &&
            (0 <= Heap[this, M.k] && Heap[this, M.k] < Length(M.a));
        ArrayHeap[Heap[this, M.a], Heap[this, M.k]] := ArrayHeap[Heap[this, M.b↩
↪], Heap[this, M.k]];
        assert Permission[this, M.k] == 1.0;
        Heap[this, M.k] := Heap[this, M.k] + 1;
    }
    //postcondition
    assert 0.0 < Permission[this, M.left] && 0.0 < Permission[this, M.end];
    assert (forall i, j: int :: (Heap[this, M.left] <= i && i < j && j <= Heap[↩
↪this, M.end]) ==>
            ArrayHeap[Heap[this, M.a],i] <= ArrayHeap[Heap[this, M.a],j]);
    //permission
    //local variables
    assert Permission[this, M.ileft] == 1.0;
    Permission[this, M.ileft] := Permission[this, M.ileft] - 1.0;
    assert Permission[this, M.iright] == 1.0;
    Permission[this, M.iright] := Permission[this, M.iright] - 1.0;
    assert Permission[this, M.k] == 1.0;
    Permission[this, M.k] := Permission[this, M.k] - 1.0;
    //takes { i : {left,..M.end} . a(i) }
    oldArrayPermission := ArrayPermission;
    havoc ArrayPermission;
    assume (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
        ArrayPermission[o,f] == oldArrayPermission[o,f] - 1.0);
    assume (forall<x> o:ArrayRef x, f : int :: !((o == Heap[this, M.a]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end]))==>
```

```
            ArrayPermission[o,f] == oldArrayPermission[o,f]);
        //takes { i : {left,..M.end} . b(i) }
        oldArrayPermission := ArrayPermission;
        havoc ArrayPermission;
        assume (forall<x> o:ArrayRef x, f : int :: (o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end])==>
            ArrayPermission[o,f] == oldArrayPermission[o,f] - 1.0);
        assume (forall<x> o:ArrayRef x, f : int :: !((o == Heap[this, M.b]) && (↩
↪Heap[this, M.left] <= f && f <= Heap[this, M.end]))==>
            ArrayPermission[o,f] == oldArrayPermission[o,f]);
        //parameters
        assert Permission[this, M.left] >= 0.5;
        Permission[this, M.left] := Permission[this, M.left] - 0.5;
        assert Permission[this, M.right] >= 0.5;
        Permission[this, M.right] := Permission[this, M.right] - 0.5;
        assert Permission[this, M.end] >= 0.5;
        Permission[this, M.end] := Permission[this, M.end] - 0.5;
     goto Done;
     Done:
   }
}
```

Listing 5.3: Translation of merge code in Boogie

This program verifies successfuly. Additionally, two tests with incorrect code and incorrect specification are performed as follows:

- **Erroneous Code**

For instance, we change the code "b[k] := a[ileft];" to "b[k] := a[iright];" in the first **if** statement. This causes the invariant "(forall i, j : (left _< i /\ i < k /\ ileft _< j /\ j < right). b[i] _< a[j])" not to be maintained and the following error is given:

Error BP5005: This loop invariant might not be maintained by the loop.

- **Erroneous Specification**

For this case, we remove the specification "takes {i : {left,..,end} . b(i)}" from the contracts of the method. This makes accessing to the locations of array b wrong and the following error is shown in the lines accessing to the array b:

Error BP5001: This assertion might not hold.

5.2 Permission Transfer Scenarios

In this section, the three scenarios of permission transfer presented in section 3.6 are translated and verified. The most critical issues are that permissions are not lost and the threads can transfer permissions. The programs are tested in three points. First, thread t0_client must have full permission on the field a and no permission on the ghost fields of two servers in the end of the thread; this is marked with test A. Second, both servers must have zero permission on field a in the end, and must have full permission on their ghost fields; these testings are marked by test B and test C in the translation. The complete translation of these scenraios to Boogie are added in Appendix A.0.

Chapter 6

Conclusion

Formal verification is the act of checking the correctness of programs with formal methods of mathematics. Many approaches have been developed to verify programs. In this thesis, we used the implicit dynamic frames methodology [7] with fractional permissions [8], which permits verification of concurrent programs. Implicit dynamic frames were developed as a variant of dynamic frames [9] with access assertions on precondition and postcondtions. This methodology is based on verification condition generation and automated theorem proving. Implicit dynamic frames was first implemented in Vericool [15]. The Chalice program verifier [19] extended this approach with a mixture of fractional permissions and infinitesimal permissions which are similar to counting permissions.

Fractional permissions were introduced by Boyland [8], who used them to check inference in a concurrent typed system. Zhao [25] implemented fractional permissions in a typed system for concurrent Java programs. However, Zhao's work does not support verifying programs with programmer-provided specification. Verifying programs

with permissions was first explored in Separation logic by Bornat *et al.* [26], Gotsman *et al.* [27], and Hobor *et al.* [28].

6.0 Contribution

In this thesis, we present a verification methodology for the multi-threaded and object-oriented Harpo/L programming language. Our methodology allows verifying programs based on their contracts. More importantly, it rules out data races. Also, it supports object invariants, rendezvous and fine-grained concurrency. This methodology is built based on implicit dynamic frames with fractional permissions. An expressive and flexible specification language is developed based on this methodology. It also permits modular specification and verification.

We have provided the encoding of HARPO/L constructs to an intermediate verification language, Boogie, based on our methodology. Several test cases are implemented with this approach. Also, we have specified and successfully verified some challenging algorithms as illustrated in this thesis.

6.1 Future Work

This thesis is developed based on language design 8 of HARPO/L [6]. In future, when adding new features to the language, their verification must be considered too. In the time of this thesis, the compiler of HARPO/L was not ready; therefore one future step is to implement the developed approach in a tool. Another direction for future research is to develop the methodology to allow verification that systems do

not deadlock. All properties verified so far are safety properties and another direction for future work is to verify liveness properties as well.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*, ser. LNCS. Springer-Verlag, 2008, vol. 5000, pp. 27–45.

[3] E. Brinksma, "Verification is experimentation!" *STTT*, vol. 3, no. 2, pp. 107–111, 2001.

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[5] K. R. M. Leino, "This is Boogie 2," Microsoft Research, Tech. Rep., 2008.

[6] T. S. Norvell, "Language design for CGRA project. design 8. [unpublished draft]," 2013.

[7] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *ECOOP*, vol. 5653, 2009, pp. 148–172.

[8] J. Boyland, "Checking interference with fractional permissions," in *SAS*, ser. LNCS, vol. 2694. Springer, 2003, pp. 55–72.

[9] I. T. Kassios, "The dynamic frames theory," *Formal Aspects of Computing*, vol. 23, no. 3, pp. 267–288, 2011.

[10] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Scienc*, 2002, pp. 55–74.

[11] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01. Springer-Verlag, 2001, pp. 1–19.

[12] I. T. Kassios, "Dynamic frames and automated verification," 2011, 2nd COST Action IC0701 Training School, Limerick, Ireland.

[13] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," *Logical Methods in Computer Science*, vol. 8, no. 3, 2012.

[14] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[15] J. Smans, B. Jacobs, and F. Piessens, "Vericool: An automatic verifier for a concurrent object-oriented language," in *FMOODS*, vol. 5051. Springer-Verlag, 2008, pp. 220–239.

[16] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[17] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 271–307, 2007.

[18] J. C. Reynolds, "Toward a grainless semantics for shared-variable concurrency," in *FSTTCS*, vol. 3328, 2004, pp. 35–48.

[19] K. R. M. Leino and P. Müller, "A basis for verifying multi-threaded programs," in *ESOP*, vol. 5502. Springer, 2009, pp. 378–393.

[20] J. T. Boyland, "Semantics of fractional permissions with nesting," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 22:1–22:33, 2010.

[21] T. S. Norvell, M. A. A. Tuhin, X. Li, and D. Zhang, "HARPO/L: A language for hardware/software codesign," in *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, 2008.

[22] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[23] X. Li, "Analysis and compilation techniques for HARPO/L." Master's thesis, Memorial Univercity of Newroundland, 2008.

[24] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Abstract read permissions: Fractional permissions without the fractions," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

[25] Y. Zhao, "Concurrency analysis based on fractional permissions," Ph.D. dissertation, University of Wisconsin at Milwaukee, 2007.

[26] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *POPL*, 2005, pp. 259–270.

[27] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local reasoning for storable locks and threads," in *APLAS*, ser. LNCS, vol. 4807. Springer Verlag, 2007, pp. 19–37.

[28] A. Hobor, A. W. Appel, and F. Z. Nardelli, "Oracle semantics for concurrent separation logic," in *ESOP*. Springer, 2008, pp. 353–367.

Appendix A

An Appendix

A.0 Translation of Permission Transfer Scenarios

- **Scenario 1**

The code and specification of this scenario is presented in listing 3.3. The translation of the code in Boogie is shown below:

```
type Ref;
type Field x;
type HeapType = <x>[Ref, Field x]x;
var Heap : HeapType;
type Perm = real;
type PermissionType = <x>[Ref, Field x] Perm;
var LockPermission : PermissionType;
type ClassName;
function dtype(Ref) returns(ClassName);

const unique C : ClassName;
const unique C.a : Field bool;
const unique C.gp1 : Field Perm;
const unique C.gp2 : Field Perm;
const unique C.p1 : Field Perm;
const unique C.p2 : Field Perm;
```

```
const unique C.p1_init : Field Perm;
const unique C.p2_init : Field Perm;
const unique C.pc1 : Field Perm;
const unique C.pc2 : Field Perm;

procedure C.C(this:Ref)
  requires dtype(this) <: 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, C.a] := Permission[this, C.a] + 1.0;
   Permission[this, C.gp1] := Permission[this, C.gp1] + 1.0;
   Permission[this, C.gp2] := Permission[this, C.gp2] + 1.0;
   assert (forall<x> r:Ref, f : Field x :: Permission[r, f] <= 1.0);
   //initialize
   Heap[this, C.a] := false;
   Heap[this, C.gp1] := 0.0;
   Heap[this, C.gp2] := 0.0;
}

procedure C.t0_client(this : Ref)
  modifies Heap;
  requires dtype(this) <: 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, C.a] := Permission[this, C.a] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.a] == false;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.a) ==> Heap[r, f↩
↪] == oldHeap[r, f]);
  while(true)
```

```
    invariant Permission[this, C.a] == 1.0;
  {
    //thread body
    Permission[this, C.pc1] := Permission[this, C.pc1] + 1.0;
    assert Permission[this, C.pc1] == 1.0;
    Heap[this, C.pc1] := 0.5;
    //-----call worker1_start
    preHeap := Heap;
    assert Permission[this, C.pc1] > 0.0;
    p1 := Heap[this, C.pc1];
    that := this;
    //precondition
    assert p1 > 0.0 && p1 < 1.0;
    assert Permission[that, C.a] >= p1;
    Permission[that, C.a] := Permission[that, C.a] - p1;
    //postcondition
    if (Permission[that, C.gp1] == 0.0)
    {
      havoc Heap_tmp;
      Heap[that, C.gp1] := Heap_tmp[that, C.gp1];
    }
    Permission[that, C.gp1] := Permission[that, C.gp1] + 0.5;
    oldHeap := Heap;
    havoc Heap;
    assume Heap[that, C.gp1] == p1;
    assume (forall <x> r:Ref, f : Field x :: !(r==that && f == C.gp1) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
    //-----call worker1_finish
    preHeap := Heap;
    that := this;
    assert Permission[this, C.pc1] > 0.0;
    p1_init := Heap[this, C.pc1];
    //precondition
    assert p1_init == Heap[that, C.gp1];
    assert Permission[that, C.gp1] >= 0.5;
    Permission[that, C.gp1] := Permission[that, C.gp1] - 0.5;
    //postcondition
    if (Permission[that, C.a] == 0.0)
    {
      havoc Heap_tmp;
      Heap[that, C.a] := Heap_tmp[that, C.a];
    }
```

```
    Permission[that, C.a] := Permission[that, C.a] + p1_init;
     //test A
    assert Permission[this, C.a] == 1.0;
    assert Permission[this, C.gp1] == 0.0;
    assert Permission[this, C.gp2] == 0.0;
    //lose local permission
    assert Permission[this, C.pc1] == 1.0;
    Permission[this, C.pc1] := Permission[this, C.pc1] - 1.0;
  }
}

procedure C.t1_server1(this : Ref)
  modifies Heap;
  requires dtype(this) <: 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, C.gp1] := Permission[this, C.gp1] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.gp1] == 0.0;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp1) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while(true)
    invariant Permission[this, C.gp1] == 1.0;
  {
    //-----worker1_start
    goto worker1_start;
    worker1_start:
    //precondition
    if (Permission[this, C.p1] == 0.0)
    {
      havoc Heap_tmp;
      Heap[this, C.p1] := Heap_tmp[this, C.p1];
    }
    Permission[this, C.p1] := Permission[this, C.p1] + 0.5;
    oldHeap := Heap;
    havoc Heap;
```

```
    assume Heap[this, C.p1] > 0.0 && Heap[this, C.p1] < 1.0;
    assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.p1) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
    if (Permission[this, C.a] == 0.0)
    {
      havoc Heap_tmp;
      Heap[this, C.a] := Heap_tmp[this, C.a];
    }
    Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p1];
    //body
    preHeap := Heap;
    assert Permission[this, C.p1] > 0.0;
    assert Permission[this, C.gp1] == 1.0;
    Heap[this, C.gp1] := Heap[this, C.p1];
    //postcondition
    assert Heap[this, C.gp1] == preHeap[this, C.p1];
    Permission[this, C.gp1] := Permission[this, C.gp1] - 0.5;
    Permission[this, C.p1] := Permission[this, C.p1] - 0.5;
    goto Done_worker1_start;
    Done_worker1_start:
    //thread code
    Permission[this, C.pc2] := Permission[this, C.pc2] + 1.0;
    assert Permission[this, C.pc2] == 1.0;
    assert Permission[this, C.gp1] > 0.0;
    Heap[this, C.pc2] := Heap[this, C.gp1] / 2;
    //-----call worker2_start
    oldHeap := Heap;
    assert Permission[this, C.pc2] > 0.0;
    p2 := Heap[this, C.pc2];
    that := this;
    //precondition
    assert p2 > 0.0 && p2 < 1.0;
    assert Permission[that, C.a] >= p2;
    Permission[that, C.a] := Permission[that, C.a] - p2;
    //postcondition
    if (Permission[that, C.gp2] == 0.0)
    {
      havoc Heap_tmp;
      Heap[that, C.gp2] := Heap_tmp[that, C.gp2];
    }
    Permission[that, C.gp2] := Permission[that, C.gp2] + 0.5;
    oldHeap := Heap;
```

```
      havoc Heap;
      assume Heap[that, C.gp2] == p2;
      assume (forall <x> r:Ref, f : Field x :: !(r==that && f == C.gp2) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
      //−−−−−call worker2_finish
      oldHeap := Heap;
      assert Permission[this, C.pc2] > 0.0;
      p2_init := Heap[this, C.pc2];
      that := this;
      //precondition
      assert p2_init == Heap[that, C.gp2];
      Permission[that, C.gp2] := Permission[that, C.gp2] − 0.5;
      //postcondition
      if (Permission[that, C.a] == 0.0)
      {
        havoc Heap_tmp;
        Heap[that, C.a] := Heap_tmp[that, C.a];
      }
      Permission[that, C.a] := Permission[that, C.a] + p2_init;
      //−−−−−worker1_finish
      goto worker1_finish;
      worker1_finish:
      //precondition
      if (Permission[this, C.p1_init] == 0.0)
      {
        havoc Heap_tmp;
        Heap[this, C.p1_init] := Heap_tmp[this, C.p1_init];
      }
      Permission[this, C.p1_init] := Permission[this, C.p1_init] + 0.5;
      if (Permission[this, C.gp1] == 0.0)
      {
        havoc Heap_tmp;
        Heap[this, C.gp1] := Heap_tmp[this, C.gp1];
      }
      Permission[this, C.gp1] := Permission[this, C.gp1] + 0.5;
      oldHeap := Heap;
      havoc Heap;
      assume Heap[this, C.p1_init] == Heap[this, C.gp1];
      assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.p1_init) ↩
↪==> Heap[r, f] == oldHeap[r, f]);
      //body
      preHeap := Heap;
```

```
      //post
      Permission[this, C.a] := Permission[this, C.a] − Heap[this, C.p1_init];
      Permission[this, C.p1_init] := Permission[this, C.p1_init] − 0.5;
      goto Done_worker1_finish;
      Done_worker1_finish:
      //test B
      assert Permission[this, C.a] == 0.0;
      assert Permission[this, C.gp1] == 1.0;
      assert Permission[this, C.gp2] == 0.0;
      //lose permission on locals
      assert Permission[this, C.pc2] == 1.0;
      Permission[this, C.pc2] := Permission[this, C.pc2] − 1.0;
  }
}

procedure C.t2_server2(this : Ref)
  modifies Heap;
  requires dtype(this) <: 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, C.gp2] := Permission[this, C.gp2] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.gp2] == 0.0;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp2) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while(true)
    invariant Permission[this, C.gp2] == 1.0;
  {
      //−−−−−worker2_start
      goto worker2_start;
      worker2_start:
      //pre
      if (Permission[this, C.p2] == 0.0)
      {
        havoc Heap_tmp;
        Heap[this, C.p2] := Heap_tmp[this, C.p2];
      }
```

```
Permission[this, C.p2] := Permission[this, C.p2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume 0.0 < Heap[this, C.p2] && Heap[this, C.p2] < 1.0;
assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.p2) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
if (Permission[this, C.a] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.a] := Heap_tmp[this, C.a];
}
Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p2];
//body
preHeap := Heap;
assert Permission[this, C.p2] > 0.0;
assert Permission[this, C.gp2] == 1.0;
Heap[this, C.gp2] := Heap[this, C.p2];
//postcondition
assert Heap[this, C.gp2] == preHeap[this, C.p2];
Permission[this, C.gp2] := Permission[this, C.gp2] - 0.5;
Permission[this, C.p2] := Permission[this, C.p2] - 0.5;
goto Done_worker2_start;
Done_worker2_start:
//--------worker2_finish
goto worker2_finish;
worker2_finish:
//precondition
if (Permission[this, C.p2_init] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.p2_init] := Heap_tmp[this, C.p2_init];
}
Permission[this, C.p2_init] := Permission[this, C.p2_init] + 0.5;
if (Permission[this, C.gp2] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.gp2] := Heap_tmp[this, C.gp2];
}
Permission[this, C.gp2] := Permission[this, C.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[this, C.p2_init] == Heap[this, C.gp2];
```

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

Listing A.0: Translation of scenario 1 in Boogie

- **Scenario 2**

The code and specification of this scenario is presented in listing 3.4. The translation of the code in Boogie is shown below:

```
type Ref;
type Field x;
type HeapType = <x>[Ref, Field x]x;
var Heap : HeapType;
type Perm = real;
type PermissionType = <x>[Ref, Field x] Perm;
var LockPermission : PermissionType;
type ClassName;
function dtype(Ref) returns(ClassName);

const unique C : ClassName;
const unique C.a : Field bool;
const unique C.gp1 : Field Perm;
const unique C.gp2 : Field Perm;
const unique C.p1 : Field Perm;
const unique C.p2 : Field Perm;
```

```
const unique C.p1_init : Field Perm;
const unique C.p2_init : Field Perm;
const unique C.po1 : Field Perm;
const unique C.po2 : Field Perm;
const unique C.p : Field Perm;
const unique C.pc1 : Field Perm;
const unique C.pc2 : Field Perm;

procedure C.C(this:Ref)
  requires dtype(this) <: 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, C.a] := Permission[this, C.a] + 1.0;
   Permission[this, C.gp1] := Permission[this, C.gp1] + 1.0;
   Permission[this, C.gp2] := Permission[this, C.gp2] + 1.0;
   assert (forall<x> r:Ref, f : Field x :: Permission[r, f] <= 1.0);
   //initialize
   Heap[this, C.a] := false;
   Heap[this, C.gp1] := 0.0;
   Heap[this, C.gp2] := 0.0;
}

procedure C.t0_client(this : Ref)
  modifies Heap;
  requires dtype(this) <: C;
{
  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, C.a] := Permission[this, C.a] + 1.0;
  oldHeap := Heap;
  havoc Heap;
```

```
  assume Heap[this, C.a] == false;
  assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.a)) ==> Heap[r, ↩
↪f] == oldHeap[r, f]);
  while(true)
    invariant Permission[this, C.a] == 1.0;
  {
    //main
    preHeap0 := Heap;
      //permission on local variables
      Permission[this, C.p] := Permission[this, C.p] + 1.0;
      Heap[this, C.p] := 0.0;
      Permission[this, C.pc1] := Permission[this, C.pc1] + 1.0;
      Heap[this, C.pc1] := 0.0;
      Permission[this, C.po1] := Permission[this, C.po1] + 1.0;
      Heap[this, C.po1] := 0.0;
      Permission[this, C.po2] := Permission[this, C.po2] + 1.0;
      Heap[this, C.po2] := 0.0;
      //code
      assert Permission[this, C.p] == 1.0;
      Heap[this, C.p] := 1.0;
      assert Permission[this, C.pc1] == 1.0 && Permission[this, C.p] > 0.0;
      Heap[this, C.pc1] := Heap[this, C.p] / 2;
      assert Permission[this, C.p] == 1.0 && Permission[this, C.pc1] > 0.0;
      Heap[this, C.p] := Heap[this, C.p] - Heap[this, C.pc1];
      //-----call worker1_start
      preHeap := Heap;
      that := this;
      assert Permission[this, C.pc1] > 0.0;
      p1 := Heap[this, C.pc1];
      //precondition
      assert p1 > 0.0 && p1 < 1.0;
      assert Permission[that, C.a] >= p1;
      Permission[this, C.a] := Permission[that, C.a] - p1;
      //postcondition
      if (Permission[that, C.gp1] == 0.0)
      {
          havoc Heap_tmp;
          Heap[that, C.gp1] := Heap_tmp[that, C.gp1];
      }
      Permission[that, C.gp1] := Permission[that, C.gp1] + 0.5;
      if (Permission[that, C.gp2] == 0.0)
      {
```

```
        havoc Heap_tmp;
        Heap[that, C.gp2] := Heap_tmp[that, C.gp2];
    }
    Permission[that, C.gp2] := Permission[that, C.gp2] + 0.5;
    oldHeap := Heap;
    havoc Heap;
    assume Heap[that, C.gp1] == p1;
    assume Heap[that, C.gp2] == p1 / 2;
    assume (forall<x> r:Ref, f : Field x :: !((r==that && f == C.gp1) || (r==↩
↪that && f == C.gp1)) ==>
      Heap[r, f] == oldHeap[r, f]);
    //-----call worker1_finish
    preHeap := Heap;
    that := this;
    assert Permission[this, C.pc1] > 0.0;
    p1_init := Heap[this, C.pc1];
    //precondition
    assert p1_init == Heap[that, C.gp1];
    assert Permission[that, C.gp1] >= 0.5;
    Permission[that, C.gp1] := Permission[that, C.gp1] - 0.5;
    //postcondition
    havoc po1;
    assume po1 == p1_init / 2;
    if (Permission[that, C.a] == 0.0)
    {
        havoc Heap_tmp;
        Heap[that, C.a] := Heap_tmp[that, C.a];
    }
    Permission[that, C.a] := Permission[that, C.a] + po1;
    assert Permission[this, C.po1] == 1.0;
    Heap[this, C.po1] := po1;
    //p := p + pr1
    assert Permission[this, C.po1] > 0.0 && Permission[this, C.p] == 1.0;
    Heap[this, C.p] := Heap[this, C.p] + Heap[this, C.po1];
    //-----call worker2_finish
    preHeap := Heap;
    that := this;
    assert Permission[this, C.gp2] > 0.0;
    assert Permission[this, C.po2] == 1.0;
    p2_init := Heap[this, C.gp2];
    //precondition
    assert p2_init == Heap[that, C.gp2];
```

```
        assert Permission[that, C.gp2] >= 0.5;
        Permission[that, C.gp2] := Permission[that, C.gp2] − 0.5;
        //postcondition
        havoc po2;
        assume po2 == p2_init;
        if (Permission[that, C.a] == 0.0)
        {
            havoc Heap_tmp;
            Heap[that, C.a] := Heap_tmp[this, C.a];
        }
        Permission[that, C.a] := Permission[that, C.a] + po2;
        assert Permission[this, C.po2] == 1.0;
        Heap[this, C.po2] := po2;
        //test A
        assert Permission[this, C.a] == 1.0;
        assert Permission[this, C.gp1] == 0.0;
        assert Permission[this, C.gp2] == 0.0;
        //lose permission on local variables
        assert Permission[this, C.p] == 1.0;
        Permission[this, C.p] := Permission[this, C.p] − 1.0;
        assert Permission[this, C.pc1] == 1.0;
        Permission[this, C.pc1] := Permission[this, C.pc1] − 1.0;
        assert Permission[this, C.po1] == 1.0;
        Permission[this, C.po1] := Permission[this, C.po1] − 1.0;
        assert Permission[this, C.po2] == 1.0;
        Permission[this, C.po2] := Permission[this, C.po2] − 1.0;
    }
}

procedure C.t1_server1(this : Ref)
    modifies Heap;
    requires dtype(this) <: C;
{
    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, C.gp1] := Permission[this, C.gp1] + 1.0;
    oldHeap := Heap;
    havoc Heap;
```

```
  assume Heap[this, C.gp1] == 0.0;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp1) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while (true)
    invariant Permission[this, C.gp1] == 1.0;
  {
      //−−−−−worker1_start
      goto worker1_start;
      worker1_start:
      //precondition
      if (Permission[this, C.p1] == 0.0)
      {
          havoc Heap_tmp;
          Heap[this, C.p1] := Heap_tmp[this, C.p1];
      }
      Permission[this, C.p1] := Permission[this, C.p1] + 0.5;
      oldHeap := Heap;
      havoc Heap;
      assume Heap[this, C.p1] > 0.0 && Heap[this, C.p1] < 1.0;
      assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p1)) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
      if (Permission[this, C.a] == 0.0)
      {
          havoc Heap_tmp;
          Heap[this, C.a] := Heap_tmp[this, C.a];
      }
      assert Permission[this, C.p1] > 0.0;
      Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p1];
      //body
      preHeap := Heap;
      assert Permission[this, C.p1] > 0.0;
      assert Permission[this, C.gp1] == 1.0;
      Heap[this, C.gp1] := Heap[this, C.p1];
      Permission[this, C.pc2] := Permission[this, C.pc2] + 1.0;
      assert Permission[this, C.pc2] == 1.0;
      assert Permission[this, C.gp1] > 0.0;
      Heap[this, C.pc2] := Heap[this, C.gp1] / 2;
      //−−−−−call worker2_start
      oldHeap2 := Heap;
      that := this;
      assert Permission[this, C.pc2] > 0.0;
      p2 := Heap[this, C.pc2];
```

```
        //precondition call worker2_start
        assert p2 > 0.0 && p2 < 1.0;
        assert Permission[that, C.a] >= p2;
        Permission[that, C.a] := Permission[that, C.a] − p2;
        //postcondition call worker2_start
        if (Permission[this, C.gp2] == 0.0)
        {
            havoc Heap_tmp;
            Heap[this, C.gp2] := Heap_tmp[this, C.gp2];
        }
        Permission[this, C.gp2] := Permission[that, C.gp2] + 0.5;
        oldHeap := Heap;
        havoc Heap;
        assume Heap[that, C.gp2] == p2;
        assume (forall<x> r:Ref, f : Field x :: (!(r==that && f == C.gp2)) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
        //postcondition worker1_start
        assert Heap[this, C.gp1] == Heap[this, C.p1];
        assert Heap[this, C.gp2] == Heap[this, C.p1]/2;
        Permission[this, C.gp2] := Permission[this, C.gp2] − 0.5;
        Permission[this, C.gp1] := Permission[this, C.gp1] − 0.5;
        Permission[this, C.p1] := Permission[this, C.p1] − 0.5;
        goto Done_worker1_start;
        Done_worker1_start:
        //−−−−−worker1_finish
        goto worker1_finish;
        worker1_finish:
        //precondition
        if (Permission[this, C.p1_init] == 0.0)
        {
            havoc Heap_tmp;
            Heap[this, C.p1_init] := Heap_tmp[this, C.p1_init];
        }
        Permission[this, C.p1_init] := Permission[this, C.p1_init] + 0.5;
        if (Permission[this, C.po1] == 0.0)
        {
            havoc Heap_tmp;
            Heap[this, C.po1] := Heap_tmp[this, C.po1];
        }
        Permission[this, C.po1] := Permission[this, C.po1] + 1.0;
        if (Permission[this, C.gp1] == 0.0)
        {
```

```
        havoc Heap_tmp;
        Heap[this, C.gp1] := Heap_tmp[this, C.gp1];
      }
      Permission[this, C.gp1] := Permission[this, C.gp1] + 0.5;
      oldHeap := Heap;
      havoc Heap;
      assume Heap[this, C.p1_init] == Heap[this, C.gp1];
      assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p1_init)) ↩
↪==> Heap[r, f] == oldHeap[r, f]);
      //body
      preHeap := Heap;
      assert Permission[this, C.po1] == 1.0 && Permission[this, C.p1_init] > 0.0;
      Heap[this, C.po1] := Heap[this, C.gp1] / 2;
      //post
      assert Heap[this, C.po1] == preHeap[this, C.p1_init] / 2;
      assert Permission[this, C.a] >= Heap[this, C.po1];
      Permission[this, C.a] := Permission[this, C.a] - Heap[this, C.po1];
      assert Permission[this, C.po1] == 1.0;
      Permission[this, C.po1] := Permission[this, C.po1] - 1.0;
      assert Permission[this, C.p1_init] >= 0.5;
      Permission[this, C.p1_init] := Permission[this, C.p1_init] - 0.5;
      goto Done_worker1_finish;
      Done_worker1_finish:
      //test B
      assert Permission[this, C.a] == 0.0;
      assert Permission[this, C.gp1] == 1.0;
      assert Permission[this, C.gp2] == 0.0;
  }
}

procedure C.t2_server2(this : Ref)
  modifies Heap;
  requires dtype(this) <: C;
{
  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, C.gp2] := Permission[this, C.gp2] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.gp2] == 0.0;
```

```
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp2) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while(true)
    invariant Permission[this, C.gp2] == 1.0;
  {
      //−−−−−worker2_start
      goto worker2_start;
      worker2_start:
      //precondition
      if (Permission[this, C.p2] == 0.0)
      {
          havoc Heap_tmp;
          Heap[this, C.p2] := Heap_tmp[this, C.p2];
      }
      Permission[this, C.p2] := Permission[this, C.p2] + 0.5;
      oldHeap := Heap;
      havoc Heap;
      assume Heap[this, C.p2] > 0.0 && Heap[this, C.p2] < 1.0;
      assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p2)) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
      if (Permission[this, C.a] == 0.0)
      {
          havoc Heap_tmp;
          Heap[this, C.a] := Heap_tmp[this, C.a];
      }
      assert Permission[this, C.p2] > 0.0;
      Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p2];
      //body
      preHeap := Heap;
      assert Permission[this, C.p2] > 0.0;
      assert Permission[this, C.gp2] == 1.0;
      Heap[this, C.gp2] := Heap[this, C.p2];
      //postcondition
      assert Heap[this, C.gp2] == preHeap[this, C.p2];
      assert Permission[this, C.gp2] >= 0.5;
      Permission[this, C.gp2] := Permission[this, C.gp2] − 0.5;
      assert Permission[this, C.p2] >= 0.5;
      Permission[this, C.p2] := Permission[this, C.p2] − 0.5;
      goto Done_worker2_start;
      Done_worker2_start:
      //−−−−−worker2_finish
      goto worker2_finish;
```

```
worker2_finish:
//precondition
if (Permission[this, C.p2_init] == 0.0)
{
    havoc Heap_tmp;
    Heap[this, C.p2_init] := Heap_tmp[this, C.p2_init];
}
Permission[this, C.p2_init] := Permission[this, C.p2_init] + 0.5;
if (Permission[this, C.po2] == 0.0)
{
    havoc Heap_tmp;
    Heap[this, C.po2] := Heap_tmp[this, C.po2];
}
Permission[this, C.po2] := Permission[this, C.po2] + 1.0;
if (Permission[this, C.gp2] == 0.0)
{
    havoc Heap_tmp;
    Heap[this, C.gp2] := Heap_tmp[this, C.gp2];
}
Permission[this, C.gp2] := Permission[this, C.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[this, C.p2_init] == Heap[this, C.gp2];
assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p2_init)) ↩
↪==> Heap[r, f] == oldHeap[r, f]);
//body
preHeap := Heap;
assert Permission[this, C.p2_init] > 0.0;
assert Permission[this, C.po2] == 1.0;
Heap[this, C.po2] := Heap[this, C.p2_init];
//post
assert Heap[this, C.po2] == preHeap[this, C.p2_init];
assert (Permission[this, C.a] >= Heap[this, C.po2]) && (Permission[this, ↩
↪C.po2] > 0.0);
Permission[this, C.a] := Permission[this, C.a] - Heap[this, C.po2];
assert Permission[this, C.po2] == 1.0;
Permission[this, C.po2] := Permission[this, C.po2] - 1.0;
assert Permission[this, C.p2_init] >= 0.5;
Permission[this, C.p2_init] := Permission[this, C.p2_init] - 0.5;
goto Done_worker2_finish;
Done_worker2_finish:
//test C
```

```
        assert Permission[this, C.a] == 0.0;
        assert Permission[this, C.gp1] == 0.0;
        assert Permission[this, C.gp2] == 1.0;
    }
}
```

Listing A.1: Translation of scenario 2 in Boogie

- **Scenario 3**

The code and specification of this scenario is presented in listing 3.5. The translation of the code in Boogie is shown below:

```
type Ref;
type Field x;
type HeapType = <x>[Ref, Field x]x;
var Heap : HeapType;
type Perm = real;
type PermissionType = <x>[Ref, Field x] Perm;
var LockPermission : PermissionType;
type ClassName;
function dtype(Ref) returns(ClassName);

const unique C : ClassName;
const unique C.a : Field bool;
const unique C.gp1 : Field Perm;
const unique C.gp2 : Field Perm;
const unique C.p1 : Field Perm;
const unique C.p2 : Field Perm;
const unique C.p1_init : Field Perm;
const unique C.p2_init : Field Perm;
const unique C.po : Field Perm;
const unique C.p : Field Perm;
const unique C.pc1 : Field Perm;
const unique C.pc2 : Field Perm;
const unique C.pr : Field Perm;

procedure C.C(this:Ref)
  requires dtype(this) <: 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, C.a] := Permission[this, C.a] + 1.0;
    Permission[this, C.gp1] := Permission[this, C.gp1] + 1.0;
    Permission[this, C.gp2] := Permission[this, C.gp2] + 1.0;
    assert (forall<x> r:Ref, f : Field x :: Permission[r, f] <= 1.0);
    //initialize
    Heap[this, C.a] := false;
    Heap[this, C.gp1] := 0.0;
    Heap[this, C.gp2] := 0.0;
}

procedure C.t0_client(this : Ref)
  modifies Heap;
  requires dtype(this) <: C;
{
  var Permission : PermissionType where
                    (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
  var oldHeap, oldHeap2, preHeap, preHeap0, Heap_tmp : HeapType;
  var p1, p1_init, po : Perm;
  var p2, p2_init : Perm;
  var that : Ref;
  //claim
  Permission[this, C.a] := Permission[this, C.a] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.a] == false;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.a) ==> Heap[r, f↩
↪] == oldHeap[r, f]);
  while (true)
    invariant Permission[this, C.a] == 1.0;
  {
      //main
      preHeap0 := Heap;
      //local variables
      Permission[this, C.p] := Permission[this, C.p] + 1.0;
      Heap[this, C.p] := 1.0;
      Permission[this, C.pc1] := Permission[this, C.pc1] + 1.0;
```

```
Heap[this, C.pc1] := 0.0;
Permission[this, C.pc2] := Permission[this, C.pc2] + 1.0;
Heap[this, C.pc2] := 0.0;
Permission[this, C.po] := Permission[this, C.po] + 1.0;
Heap[this, C.po] := 0.0;
//code
assert Permission[this, C.pc1] == 1.0 && Permission[this, C.p] > 0.0;
Heap[this, C.pc1] := Heap[this, C.p] / 2;
assert Permission[this, C.p] == 1.0 && Permission[this, C.pc1] > 0.0;
Heap[this, C.p] := Heap[this, C.p] - Heap[this, C.pc1];
assert Permission[this, C.pc2] == 1.0 && Permission[this, C.p] > 0.0;
Heap[this, C.pc2] := Heap[this, C.p] / 2;
assert Permission[this, C.p] == 1.0 && Permission[this, C.pc2] > 0.0;
Heap[this, C.p] := Heap[this, C.p] - Heap[this, C.pc2];
//-----call worker1_start
preHeap := Heap;
that := this;
assert Permission[this, C.pc1] > 0.0;
p1 := Heap[this, C.pc1];
//precondition
assert p1 > 0.0 && p1 < 1.0;
assert Permission[that, C.a] >= p1;
Permission[that, C.a] := Permission[that, C.a] - p1;
//postcondition
Permission[that, C.gp1] := Permission[that, C.gp1] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[that, C.gp1] == p1;
assume (forall<x> r:Ref, f : Field x :: !(r==that && f == C.gp1) ==>
  Heap[r, f] == oldHeap[r, f]);
//-----call worker2_start
preHeap := Heap;
that := this;
assert Permission[this, C.pc2] > 0.0;
p2 := Heap[this, C.pc2];
//precondition
assert p2 > 0.0 && p2 < 1.0;
assert Permission[that, C.a] >= p2;
Permission[that, C.a] := Permission[that, C.a] - p2;
//postcondition
if (Permission[that, C.gp2] == 0.0)
{
```

```
  havoc Heap_tmp;
  Heap[that, C.gp2] := Heap_tmp[that, C.gp2];
}
Permission[that, C.gp2] := Permission[that, C.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[that, C.gp2] == p2;
assume (forall<x> r:Ref, f : Field x :: (!(r==that && f == C.gp2)) ==>
  Heap[r, f] == oldHeap[r, f]);
//-----call worker2_finish
preHeap := Heap;
that := this;
assert Permission[this, C.pc1] > 0.0;
assert Permission[this, C.pc2] > 0.0;
assert Permission[this, C.po] == 1.0;
p1_init := Heap[this, C.pc1];
p2_init := Heap[this, C.pc2];
//precondition
assert p1_init == Heap[that, C.gp1];
assert p2_init == Heap[that, C.gp2];
assert Permission[that, C.gp1] >= 0.5;
Permission[that, C.gp1] := Permission[that, C.gp1] - 0.5;
assert Permission[that, C.gp2] >= 0.5;
Permission[that, C.gp2] := Permission[that, C.gp2] - 0.5;
//postcondition
havoc po;
assume po == p1_init + p2_init;
if (Permission[that, C.a] == 0.0)
{
  havoc Heap_tmp;
  Heap[that, C.a] := Heap_tmp[that, C.a];
}
Permission[that, C.a] := Permission[that, C.a] + po;
assert Permission[this, C.po] == 1.0;
Heap[this, C.po] := po;
//test A
assert Permission[this, C.a] == 1.0;
assert Permission[this, C.gp1] == 0.0;
assert Permission[this, C.gp2] == 0.0;
//lose permission on local variables
assert Permission[this, C.p] == 1.0;
Permission[this, C.p] := Permission[this, C.p] - 1.0;
```

```
      assert Permission[this, C.pc1] == 1.0;
      Permission[this, C.pc1] := Permission[this, C.pc1] − 1.0;
      assert Permission[this, C.pc2] == 1.0;
      Permission[this, C.pc2] := Permission[this, C.pc2] − 1.0;
      assert Permission[this, C.po] == 1.0;
      Permission[this, C.po] := Permission[this, C.po] − 1.0;
  }
}

procedure C.t1_server1(this : Ref)
  modifies Heap;
  requires dtype(this) <: C;
{
  var Permission : PermissionType where
                    (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
  var oldHeap, preHeap, Heap_tmp : HeapType;
  //claim
  Permission[this, C.gp1] := Permission[this, C.gp1] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.gp1] == 0.0;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp1) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while (true)
    invariant Permission[this, C.gp1] == 1.0;
  {
      //−−−−−worker1_start
      goto worker1_start;
      worker1_start:
      //precondition worker1_start
      if (Permission[this, C.p1] == 0.0)
      {
        havoc Heap_tmp;
        Heap[this, C.p1] := Heap_tmp[this, C.p1];
      }
      Permission[this, C.p1] := Permission[this, C.p1] + 0.5;
      oldHeap := Heap;
      havoc Heap;
      assume Heap[this, C.p1] > 0.0 && Heap[this, C.p1] < 1.0;
      assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p1)) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
      if (Permission[this, C.a] == 0.0)
```

```
{
  havoc Heap_tmp;
  Heap[this, C.a] := Heap_tmp[this, C.a];
}
assert Permission[this, C.p1] > 0.0;
Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p1];
//body worker1_start
preHeap := Heap;
assert Permission[this, C.p1] > 0.0;
assert Permission[this, C.gp1] == 1.0;
Heap[this, C.gp1] := Heap[this, C.p1];
//postcondition worker1_start
assert Heap[this, C.gp1] == preHeap[this, C.p1];
assert Permission[this, C.gp1] >= 0.5;
Permission[this, C.gp1] := Permission[this, C.gp1] - 0.5;
assert Permission[this, C.p1] >= 0.5;
Permission[this, C.p1] := Permission[this, C.p1] - 0.5;
goto Done_worker1_start;
Done_worker1_start:
//-----worker1_finish
goto worker1_finish;
worker1_finish:
//precondition
if (Permission[this, C.p1_init] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.p1_init] := Heap_tmp[this, C.p1_init];
}
Permission[this, C.p1_init] := Permission[this, C.p1_init] + 0.5;
if (Permission[this, C.gp1] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.gp1] := Heap_tmp[this, C.gp1];
}
Permission[this, C.gp1] := Permission[this, C.gp1] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[this, C.p1_init] == Heap[this, C.gp1];
assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p1_init)) ↩
↪==> Heap[r, f] == oldHeap[r, f]);
//body worker1_finish
preHeap := Heap;
```

```
      //postcondition worker1_finish
      assert Permission[this, C.a] >= Heap[this, C.p1_init];
      assert Permission[this, C.p1_init] > 0.0;
      Permission[this, C.a] := Permission[this, C.a] − Heap[this, C.p1_init];
      assert Permission[this, C.p1_init] >= 0.5;
      Permission[this, C.p1_init] := Permission[this, C.p1_init] − 0.5;
      goto Done_worker1_finish;
      Done_worker1_finish:
      //test B
      assert Permission[this, C.a] == 0.0;
      assert Permission[this, C.gp1] == 1.0;
      assert Permission[this, C.gp2] == 0.0;
  }
}

procedure C.t2_server2(this : Ref)
  modifies Heap;
  requires dtype(this) <: C;
{
  var Permission : PermissionType where
                    (forall<x> r:Ref, f : Field x :: Permission[r, f] == 0.0);
  var oldHeap, oldHeap2, Heap_tmp, preHeap, preHeap2 : HeapType;
  var p1, p1_init : Perm;
  var that : Ref;
  //claim
  Permission[this, C.gp2] := Permission[this, C.gp2] + 1.0;
  oldHeap := Heap;
  havoc Heap;
  assume Heap[this, C.gp2] == 0.0;
  assume (forall <x> r:Ref, f : Field x :: !(r==this && f == C.gp2) ==> Heap[r↩
↪, f] == oldHeap[r, f]);
  while(true)
    invariant Permission[this, C.gp2] == 1.0;
  {
      //−−−−−worker2_start
      goto worker2_start;
      worker2_start:
      //precondition
      if (Permission[this, C.p2] == 0.0)
      {
        havoc Heap_tmp;
        Heap[this, C.p2] := Heap_tmp[this, C.p2];
```

```
        }
        Permission[this, C.p2] := Permission[this, C.p2] + 0.5;
        oldHeap := Heap;
        havoc Heap;
        assume Heap[this, C.p2] > 0.0 && Heap[this, C.p2] < 1.0;
        assume (forall<x> r:Ref, f : Field x :: (!(r==this && f == C.p2)) ==> ↩
↪Heap[r, f] == oldHeap[r, f]);
        if (Permission[this, C.a] == 0.0)
        {
          havoc Heap_tmp;
          Heap[this, C.a] := Heap_tmp[this, C.a];
        }
        assert Permission[this, C.p2] > 0.0;
        Permission[this, C.a] := Permission[this, C.a] + Heap[this, C.p2];
        //body
        preHeap := Heap;
        assert Permission[this, C.p2] > 0.0;
        assert Permission[this, C.gp2] == 1.0;
        Heap[this, C.gp2] := Heap[this, C.p2];
        //postcondition
        assert Heap[this, C.gp2] == preHeap[this, C.p2];
        assert Permission[this, C.gp2] >= 0.5;
        Permission[this, C.gp2] := Permission[this, C.gp2] - 0.5;
        assert Permission[this, C.p2] >= 0.5;
        Permission[this, C.p2] := Permission[this, C.p2] - 0.5;
        goto Done_worker2_start;
        Done_worker2_start:
        //-----worker2_finish
        goto worker2_finish;
        worker2_finish:
        //precondition
        if (Permission[this, C.p1_init] == 0.0)
        {
          havoc Heap_tmp;
          Heap[this, C.p1_init] := Heap_tmp[this, C.p1_init];
        }
        Permission[this, C.p1_init] := Permission[this, C.p1_init] + 0.5;
        if (Permission[this, C.p2_init] == 0.0)
        {
          havoc Heap_tmp;
          Heap[this, C.p2_init] := Heap_tmp[this, C.p2_init];
        }
```

```
Permission[this, C.p2_init] := Permission[this, C.p2_init] + 0.5;
if (Permission[this, C.po] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.po] := Heap_tmp[this, C.po];
}
Permission[this, C.po] := Permission[this, C.po] + 1.0;
if (Permission[this, C.gp1] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.gp1] := Heap_tmp[this, C.gp1];
}
Permission[this, C.gp1] := Permission[this, C.gp1] + 0.5;
if (Permission[this, C.gp2] == 0.0)
{
  havoc Heap_tmp;
  Heap[this, C.gp2] := Heap_tmp[this, C.gp2];
}
Permission[this, C.gp2] := Permission[this, C.gp2] + 0.5;
oldHeap := Heap;
havoc Heap;
assume Heap[this, C.p2_init] == Heap[this, C.gp2];
assume Heap[this, C.p1_init] == Heap[this, C.gp1];
assume (forall<x> r:Ref, f : Field x :: (!((r==this && f == C.p2_init) || (↩
↪r==this && f == C.p1_init)) ==>
  Heap[r, f] == oldHeap[r, f]));
//body
preHeap := Heap;
//----------call worker1_finish
preHeap2 := Heap;
that := this;
assert Permission[this, C.p1_init] > 0.0;
p1_init := Heap[this, C.p1_init];
//precondition worker1_finish
assert p1_init == Heap[that, C.gp1];
assert Permission[that, C.gp1] >= 0.5;
Permission[that, C.gp1] := Permission[that, C.gp1] - 0.5;
//postcondition worker1_finish
if (Permission[that, C.a] == 0.0)
{
  havoc Heap_tmp;
  Heap[that, C.a] := Heap_tmp[that, C.a];
```

```
        }
        Permission[that, C.a] := Permission[that, C.a] + p1_init;
        //po := p1_init + p2_init
        assert Permission[this, C.po] == 1.0 && Permission[this, C.p1_init] > 0.0 ↩
↪&& Permission[this, C.p2_init] > 0.0;
        Heap[this, C.po] := Heap[this, C.p1_init] + Heap[this, C.p2_init];
        //postcondition worker2_finish
        assert Heap[this, C.po] == preHeap[this, C.p1_init] + preHeap[this, ↩
↪C.p2_init];
        assert Permission[this, C.a] >= Heap[this, C.po];
        Permission[this, C.a] := Permission[this, C.a] - Heap[this, C.po];
        assert Permission[this, C.po] == 1.0;
        Permission[this, C.po] := Permission[this, C.po] - 1.0;
        assert Permission[this, C.p1_init] >= 0.5;
        Permission[this, C.p1_init] := Permission[this, C.p1_init] - 0.5;
        assert Permission[this, C.p2_init] >= 0.5;
        Permission[this, C.p2_init] := Permission[this, C.p2_init] - 0.5;
        goto Done_worker2_finish;
        Done_worker2_finish:
        //test C
        assert Permission[this, C.a] == 0.0;
        assert Permission[this, C.gp1] == 0.0;
        assert Permission[this, C.gp2] == 1.0;
    }
}
```

Listing A.2: Translation of scenario 3 in Boogie