MONASH University

Automatic Inference of Symbolic Permissions for Single-threaded Java Programs

Ayesha Sadiq
(MSCS, B.Sc.(Hons) CS)

A thesis
submitted for the degree of Doctor of Philosophy (PhDComp 0190) at
Monash University in 2019
Faculty of Information Technology

Copyright notice

Abstract

In mainstream programming languages such as Java, a common way to enable concurrency is to manually introduce explicit concurrency constructs such as multi-threading. In multi-threaded programs, managing synchronization between threads is a complicated and challenging task for programmers due to thread interleaving and heap interference that leads to common concurrency problems such as data races and deadlocks. Access permissions are abstract capabilities that model read and write effects of a reference on a referenced object, in the presence or absence of aliases. With these considerations in mind, access permission-based dependencies (contracts) have been investigated, as an alternative approach to verifying the correctness of the already concurrent programs and to exploit the implicit concurrency present in single-threaded (sequential) programs, without using explicit concurrency constructs such as multi-threading. However, significant annotation overhead can arise from manually adding permission-based specifications in the source program, diminishing the effectiveness of existing permission-based approaches. Moreover, given the intricacies in creating permission-based contracts, it is very likely for a programmer to omit important dependencies or to create misspelled specifications that may again lead to verification problems. Therefore, the aim of this research is to free the programmers from the annotation overhead, by automatically inferring access permission contracts from the source code of a program.

In this thesis, we present a permission inference approach that performs inter-procedural static analysis of the source code to automatically reveal read/write accesses for sequential Java programs and maps them in the form of access permission contracts. Moreover, the core functionalities of the inference approach are implemented as a prototype tool, `Sip4J`, along with its integration with, and extension of a permission-based model-checking tool, Pulse, to automatically verify the correctness of the inferred specifications and to reason about their concurrent behaviors. Our evaluation on widely-used benchmark and realistic Java applications gives strong evidence of the correctness of the inferred annotations and their effectiveness in automatically enabling concurrency for sequential programs and identifying program properties such null references and code reachability.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

PrintName: Ayesha Sadiq

Date: 22nd, October 2019

Acknowledgements

First of all, I would like to pay my sincere gratitude to my Allah Almighty who has bestowed His countless blessings upon me and supported me spiritually and worldly throughout this endeavor. I am thankful to my great family: my parents, my husband and my kids who have been affectionate and passionate about my successes and my life as a whole. I would like to thank my sisters and brother for showing compassion throughout my candidature.

I am grateful to the Government of Australia and Monash University for giving me an opportunity for higher studies and making my dreams come true. I acknowledge that this research was conducted under an Australian Government Endeavour Leadership Program (ELP) award. The major part of the award activities (study and living allowances) was funded by the Department of Education and Training and a top-up scholarship was provided by the Monash University, Faculty of Information Technology, to fill the gap in the tuition fee. I would like to extend my thanks to the administration and management of Monash University for providing students with the fabulous and outstanding research facilities.

I would like to express my sincere gratitude to my main supervisor Dr. Chris Ling, for giving me the opportunity to work under his supervision and to all of my supervisors, Dr. Yuan-Fang Li, Dr. Li Li and Dr. Ijaz Ahmed for their continuous research and motivational support for the Ph.D. study. Their guidance helped me throughout the candidature and writing up of this thesis. Without their support, it would not be possible to conduct this research and fulfill the requirements of this award.

Besides my supervisors, I would like to say a huge thanks to rest of my thesis committee: Dr. Maria Indrawan, Dr. Henry Linger, Dr. Guido Tack and Dr. Aldeida Altei for appreciating my research and for giving insightful comments which incepted me to widen my research from different perspectives.

My sincere thanks to everyone involved from FIT Graduate Research committee (especially, Helen Cridland, Allison Mitchell and Julie Holden), Monash Graduate Research office, and the whole ELP team for their amiable behavior and support services throughout the candidature. I am thankful to Mr. Robert Gray for proofreading my thesis chapters and indeed many thanks to my seniors, Dr. Masita, Dr. Yuri, for their encouragement and moral support. In particular, to my fellow colleagues and friends Ekjyot Kaur, Shabnam Kasra and Sameen Maruf for sharing their thoughts and providing feedback on my presentations.

Last but not least, I would like to thank all my family friends for their social support throughout writing this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
API	Application Programming Interfaces
BOT	Barcelona OpenMP Tasks Suite
CSL	Concurrent Separation Logic
CMEs	Concurrent Modification Exceptions
CMU	Carnegie Mellon University
CTL	Computation Tree Logic
DFA	Data Flow Analysis
evmdd	edge-valued multi-way decision diagrams
FIT	Faculty of Information Technology
FSM	Finite State Machine
FFT	Fast-Fourier Transform
HPCC	High Performance Cluster Computing
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JDK	Java Development Kit
JML	Java Modelling Language
JSL	Java Specification Language
LTL	Linear Temporal Logic
MTTS	Multi Task Threaded Server
NSL	New Specification Language.
POSIX	Portable Operating System Interface
PSM	Permission Sharing Model
PSAM	Permission Sharing and Accounting Model
RGA	Reachability Graph Analysis
SAL	Symbolic Analysis Laboratory
SFEA	Symbolic Forward Execution Analysis
SLOC	Source Lines of Code
VCs	Verification Conditions

Chapter 1

Introduction

1.1 Thesis Goal

Multi-core architectures are ubiquitous and have become the norm due to the parallel execution power made available by the multi-core processors in them. However, the software industry has not yet been able to fully exploit the performance boost of multi-core systems, principally due to inherent limitations of imperative and object-oriented programming languages such as Java. As Herb (Sutter and Larus, 2005) famously stated "`The free lunch is over`" which means the applications written in these languages can no longer benefit from the free ride (rapid performance improvements) unless programmers redesign the applications or otherwise exploit the potential concurrency present in the system.

In imperative programs, there exist implicit dependencies between code and program states which means two methods might be dependent on the same mutable state without the caller knowing about it. This information is not revealed to the compiler and alternatively, runtime system follows the execution order in which program is written, i.e., sequential and cannot exploit potential concurrency present in these systems. Enabling concurrency for imperative and object-oriented programs has become one of the grand challenges for the IT industry today[1].

In these languages, programmers introduce concurrency manually by using explicit concurrency constructs, e.g., multi-threading-related classes such as `Thread`, `Runnable` in Java. Unfortunately, the traditional multi-threading paradigm frequently results in deadlocks or unwarranted race conditions, due to thread-synchronization and heap-interference, that are hard to debug. Therefore, the correctness of such applications has always been at stake. The main problem when programmers deal with concurrency in the imperative and object-oriented languages is the shared states and the side effects (undesirable behavior due to a change in some value outside the method scope), that methods can produce on each other when executed in parallel.

The study of literature shows that symbolic permissions (Bierhoff and Aldrich, 2007; Beckman et al., 2008), simply called **access permission**, is a novel abstraction that combines the read, write and aliasing information of a referenced object. Symbolic permissions represent and track the object's accesses through the system using symbolic values such as

[1] UK Computing Research Committee, Grand Challenges in Computing Research, GCCR'08 Final Report, GC7: Journeys in Non-Classical Computation. http://www.ukcrc.org.uk/grand-challenge/.

`unique` or `immutable`.

A group of CMU researchers led by Jonathan Aldrich manually wrote permission-based typestate specifications on a number of Java APIs to model and reason about the correctness of usage protocols in sequential and concurrent programs (Bierhoff and Aldrich, 2007, 2008; Beckman, 2009; Bierhoff et al., 2009a) etc., and further parallelize the execution of these programs in a typestate-oriented programming paradigm Plaid (Aldrich et al., 2011, 2012) based on access permissions. Further, having the permission support in the Plaid infrastructure, the group worked in a joint research project, Æminium (Stork et al., 2009, 2014) and presented a by-default concurrent programming paradigm, with its own formal language and runtime engine, to parallelize execution of sequential programs based on access permissions.

Similarly, permission-based specifications have been used in many formal approaches to address issues related to safe concurrency, security and verification of functional and domain-specific properties (Leino and Müller, 2009; Leino et al., 2009; Wickerson et al., 2010; Jacobs et al., 2011; Siminiceanu et al., 2012; Heule et al., 2013; Cataño et al., 2014; Boyland et al., 2014; Juhasz et al., 2014; Müller et al., 2017; Jacobs et al., 2018) and many more. Furthermore, inference of access notations in the form of fractional and quantified permissions has recently been investigated by Peter Müller and his colleagues (Ferrara and Müller, 2012; Dohrau et al., 2018) to verify class-based concurrent programs based on the abstract interpretations (Cousot, 1996). A detailed review of the state-of-the-art permission-based program verification and parallelization approaches and their research challenges are given in Chapter 6.

Unfortunately, in order to benefit from access permissions, in all the relevant state-of-the-art approaches, programmers need to manually add appropriate permission-based specifications (e.g., annotations) as dependency information in the program. Not only do programmers need to spend time becoming familiarised with the completely new specification language(s) and runtime systems, they also need to manually identify and add specifications at the code level which is laborious and error-prone. Given the intricacies in creating these constructs, it is very likely for a programmer to omit important dependencies or create misspelled specifications that may again lead to problems such as race-conditions or deadlocks due to the wrong specifications. Moreover, handling fractional permissions is challenging and creates a complex reasoning overhead associated with tracking the concrete values in the system. These issues have hindered the wider adoption of permission-based verification and parallelization approaches. Ideally, it would be much easier and more effective for a programmer to not to be concerned with identifying and manually specifying permission-based annotations in the program, while still being able to exploit the benefits offered by such annotations.

Therefore, the aim of this research is to resolve the aforementioned issues by introducing to the community a novel approach that automatically infers symbolic permissions for the mainstream programming language Java. **The goal is to free programmers from the annotation overhead for manually adding permission-based dependencies in the program, thereby solving the common problem faced by the existing access permission-based approaches.** The research presented in this thesis is built on our initial idea (Sadiq et al., 2016) to extract the access permission rights from the source code of Java

programs. The work presented in this thesis provides a more comprehensive approach and a fully automated framework to infer and verify the inferred specifications for sequential Java programs.

1.2 A Motivating Example

In the real world, most of the existing applications are still being written in sequential programming paradigms, without using multi-threading, which cannot benefit from the characteristics of multi-core machines. In order to benefit from modern multi-core systems, there is a need to convert traditional sequential programs to parallel programs, so as to improve the execution time of these programs and to free programmers from the low-level ordering reasoning overhead about thread synchronization.

Unfortunately, in imperative languages such as Java, because of the implicit dependencies between the code and shared states, methods dependant on the same mutable objects do not specify their side effects to each other. It is hence non-trivial for programmers to manually parallelize sequential programs without the fear of data races. Let us take Listing 1.1 as an example.

Listing 1.1 illustrates a sequential Java program with three user-defined classes `ObjectClass`, `ArrayCollection` and `Client`. These three classes contain eight methods and access two shared collection objects (i.e., `array1` and `array2`). As an example, the `ObjectClass` composes the `Client` class and manipulates its member `data` using method `manipulateObjects()`. To benefit from the modern multi-core facilities, certain methods of this sample program can be executed in parallel so as to improve the overall performance. Indeed, if given two methods do not write the same object at the same time, these two methods could be parallelized. For example, methods `computeState(array1)`, `computeState(array2)` and `manipulateObjects()` could be potentially parallelized since they do not manipulate (write) on the same shared objects. Oppositely, methods such as `incrColl(array2)` and `tidyUpColl(array2)` cannot be parallelized as they modify the same object.

Unfortunately, manually identifying and tracking object accesses and the order in which these accesses are made is a laborious and error-prone task for programmers. It is very likely for a programmer to omit important dependencies or identify the wrong dependencies. The situation becomes worse in the case of unrestricted aliasing in the program, the hallmark feature of imperative programming models (Bierhoff et al., 2009b).

Listing 1.1: A sample Java program.

```
1 class ArrayCollection{
2  public Integer[] array1;
3  ArrayCollection(){
4  array1 = new Integer[10];
5  for(int i = 0; i < array1.length; i++)
6   array1[i] = (int)(Math.random() * 10);
7  }
8  public void printColl(Integer[] coll) {
9   for(int i = 0; i < coll.length; i++){
10    System.out.println(" "+coll[i]);}
11  }
12  public  void incrColl(Integer [] coll) {
13   for (int i = 0; i < this.array1.length; i++){
```

```
14      this.array1[i] = this.array1[i] + i; }
15    for (int j = 0; j < coll.length; j++){
16      coll[j] = coll[j] + j; }
17   }
18   public static boolean isSorted(Integer[] coll) {
19   boolean flag = false;
20   int j = 0;
21   for (int i = 0; i < coll.length && j < coll.length; i++,j++){
22    if(coll[i] > coll[j])
23     flag = true;
24    else
25     flag = false;
26    }
27    return flag;
28   }
29   public static Integer findMax(Integer[] coll) {
30    int max;
31    max = coll[0];
32    for (int i = 1; i < coll.length; i++){
33     if(coll[i] > max)
34       max = coll[i];}
35    return max;
36   }
37   public void computeStat(Integer[] coll){
38    printColl(coll);
39    System.out.println("Sorted = "+isSorted(coll));
40    System.out.println("Max = "+findMax(coll));
41   }
42   public void tidyupColls(Integer[] coll){
43      this.array1 = null;
44      coll = null;}
45 }
46 class ObjectClass{
47   public Integer[] array2;
48   public  Client x,y,z,w;
49   ObjectClass(){
50    array2 = new Integer[10];
51    for(int i = 0; i < array2.length; i++){
52     array2[i] = (int)(Math.random() * 10);
53    }
54    x = new Client(); y = new Client(); z = new Client(); w = new Client();
55   }
56   public void manipulateObjects(Client p1, Client p2){
57     x = p1;
58     Client t = x;
59     y = t;
60     x.data = 10;
61     System.out.println("z.data = "+p2.data);}
62 }
63 class Client{
64   Integer data = 100;
65   public static void main(String[] a) {
66    ArrayCollection obj1 = new ArrayCollection();
67    ObjectClass obj2 = new ObjectClass();
68    obj1.incrColl(obj2.array2);
69    obj1.computeStat(obj1.array1);
70    obj1.computeStat(obj2.array2);
71    obj1.tidyupColls(obj2.array2);
72    obj2.manipulateObjects(obj2.w, obj2.z);
73   }
74 }
```

Indeed, analysis of method `manipulateObjects()` shows that it accesses four objects (`x, y, z, w`) as its data members. Explicitly, it mutates only one object i.e. `x` by writing on its `data` field but actually, it mutates objects `y` and `w` as well due to aliasing of the objects in this method. These alias variables will create side effects for other methods accessing the same objects when executed in parallel. Moreover, as the program size and complexity increases, it becomes non-trivial for a programmer to identify the implicit dependencies between the different program parts by just looking at the source code.

To exploit the potential concurrency present in a Java program, every method should either avoid side effects or should explicitly mention them. This information can be used

to compute the data dependencies at different levels of granularity within the code and parallelize execution of the program to the extent permitted by these dependencies. Therefore, a mechanism is required that can express the mutability and aliasing between shared objects that can mitigate the undesirable effects to other methods in an expressive way, while also allowing a safe execution order within the methods. The objective is to free programmers from manually identifying and tracking the implicit dependencies present at the code level.

Access permissions provides a flexible control mechanism to track all the references to a particular object and update state changes to all such references Furthermore, permission-based specifications can express the implicit dependencies present in the system, while making them explicit and pose their own ordering constraints, Access permission is hence suitable for characterizing the way a shared resource is accessed by multiple references. Therefore, access permissions can be used to perform method's operations in a non-interfering manner and parallelize code without using the low-level concurrency and ordering constraints.

1.3 Research Questions and Contributions

The main research question of this thesis is:

How can we avoid the annotation overhead for adding permission-based specifications in a sequential program to help enable potential concurrency present in it?

The objective is to free programmers from the annotation overhead for manually adding permission-based implicit dependencies in the program.

The thesis research question is divided into sub-questions as follows:

RQ1: What technique is needed to automatically extract permission-based implicit dependencies from the source code of a program?

The objective is to reveal the implicit dependencies present between the code and program states in the form of access permissions.

RQ2: How can we validate the correctness of the inferred specifications?

The objective is to automatically identify the errors (missing or misspelled) in the inferred specifications and to ensure that the specifications are semantically correct.

RQ3: How can we evaluate the effectiveness of the inferred specifications?

The objective is to demonstrate the effectiveness of the permission-based specifications in

enabling concurrency from a sequential program.

RQ4: How effective is the permission inference technique itself?

The objective is to evaluate the effectiveness of the inference technique in-terms of avoiding the annotation overhead and its scalability for realistic Java programs.

RQ5: How efficient is the permission inference technique?

The objective is to evaluate the performance of the inference mechanism in terms of its time efficiency to analyze and generate the permission-based specifications from the source code.

Research Contributions

The overall contribution of this thesis is the automatic inference of access permission rights in the form of symbolic permissions for the single-threaded (sequential) programs written in a highly popular language Java. To the best of our knowledge, our work is the first attempt of its nature to infer access permissions at a higher level of abstraction, without using any intermediate representations or method-level specifications and for the mainstream programming language.

The main contributions are as follows:

Methodology: The first contribution is a permission inference approach, that reveals implicit dependencies present between the code and the shared states in a un-annotated sequential Java program, in the form of access permissions (to answer **RQ1**).
The permission inference approach performs static analysis of the source program to identify and track the object's accesses as data-flow and alias-flow information at the method level. The approach performs flow-insensitive analysis of the source code which does not take into account the control flow of the program. Our objective is to find out how the global states of a program are affected by the execution of methods and to identify the method's side effects. The analysis is based on the Abstract Syntax Tree (AST) of the program code. The approach follows a set of pre-defined syntactic and permission inference rules to support the analysis and to generate five types of access permissions (`unique, full, pure, share and immutable`) for the objects accessed at the method level. The permission contracts are then generated, as pre- and post-permissions on the referenced objects to compute dependencies between methods.
The technique to infer access permissions in this work, although focused on Java language only, should also be applicable to other object-oriented programming languages such as .Net, C#. Further, the application of flow-insensitive analysis to identify methods's side-effects and to discover their concurrent execution outweighs the approaches that are based on the

flow-sensitive analysis and that does not always increase precision of the information discovered and in most cases, flow-insensitivity is used as an approximation of flow-sensitivity and is justified as a tradeoff that must be tolerated for achieving scalability (Khedker et al., 2009; Hardekopf and Lin, 2011).

Implementation: We designed a permission inference framework and implemented the inference approach as a prototype tool called Sip4J. Additionally, the core functionalities of Sip4J are further integrated into Eclipse as a plugin. We integrated and extended the Pulse tool, a model checking verification tool[2], as a part of the framework to automate the access permission checking mechanism and to reason about their concurrent behavior (to answer **RQ2 & RQ3**).

The implementation of `Sip4J` framework has been published as an open source project at https://github.com/Sip4J/Sip4J and a demo video of the tool can be found at https://youtu.be/RjMTIxlhHTg.

The `Sip4J` permission inference framework automatically infers access permissions for an un-annotated sequential Java program at the object's (class) field level. It automatically generates Plural[3] specifications (access permission contracts with typestate information at the method level) for the same program. The inferred specifications can be used by the users of Plural and Pulse (Siminiceanu et al., 2012) itself to verify program behavior and by other permission-based parallelization approaches Plaid (Aldrich et al., 2012) and Æminium (Stork et al., 2014), to perform their intended tasks without incurring extra work on programmers. Further, the `Sip4J` framework performs a comprehensive concurrency analysis of the input program by analyzing permission-based side effects at the method level. The `Sip4J` concurrency analysis can be employed to parallelize the execution of sequential programs written in the mainstream programming languages, without the fear of data races.

Evaluation: We evaluated Sip4J and the inferred specifications on widely used benchmark suites and real-world Java applications to demonstrate their efficacy and effectiveness (to answer **RQ4 & RQ5**).

The experimental results have shown that Sip4J is indeed capable of inferring the required or safe permissions for all the methods in the input program in a correct and efficient way. Further, we empirically evaluated the proposed technique itself in terms of its scalability and effectiveness analysis to avoid the annotation overhead that existing permission-based approaches pose to programmers. Furthermore, with additional experiments, we have shown that the inferred permissions are not only useful for enabling concurrency for sequential Java programs. It is also able to perform the code reachability analysis of the underlying program and discover some of the syntactical errors in a program such as null pointer references without actually compiling and executing the

[2] http://aeminium.dei.uc.pt/index.php/ToolsAndDownloads
[3] https://code.google.com/archive/p/pluralism/

program.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 introduces different categories of access permission sharing models and provides background information on symbolic permissions. Further, it briefly explains the permission-based verification and parallelization approaches closely related to Java as we borrow the syntax from and perform the evaluation of the inferred specifications using some of them. The following chapters focus on permission inference approach and its evaluation:

Chapter 3 lays the theoretical background of the permission inference approach and the set of graph abstractions and syntactic rules to support the analysis. It elaborates the inference approach using a motivating example. Further, it explains and elaborates on the automated permission checking mechanism through the permission-based model checking tool Pulse.

Chapter 4 discusses the detailed design and implementation of the permission inference framework as a prototype Sip4J tool, along with its integration with and extension of the Pulse tool. It further elaborates on the implications and limitations of the `Sip4J` framework and its analysis in general, and on the potential improvements to the Sip4J.

Chapter 5 presents the evaluation of the Sip4J implementation. The evaluation is based on several case studies from four benchmark suites and realistic Java applications that have been widely used in the research community to evaluate program verification and parallelization approaches. This chapter explains the results of the experiments based on the pre-defined criteria for evaluation. It further elaborates on our experience with generating and verifying the correctness of the inferred specification automatically and manually. Finally, it summarizes the results of the evaluation to show the efficacy and effectiveness of the proposed framework.

Chapter 6 provides a study of the state-of-the-art in permission-based verification and parallelization approaches, comparing and contrasting the existing approaches based on the pre-defined criteria, followed by an insight into the usage of permission-based specifications in the existing approaches and the research challenges.

Chapter 7 provides a reflection of the work presented in this thesis, on automatically inferring permission-based specifications from the source program and their implications. Finally, it concludes the thesis by discussing how access permission inference can be improved and proposes some ideas for future work.

Chapter 2

Background

This chapter briefly introduces different categories of permission sharing models and how different access permissions can co-exist with each other at the method level. It further explains the access permission splitting and joining rules and how access permission contracts can be written following the Design by Contract principle. As discussed previously, the aim is this research is to automatically generate the access permissions for Java programs. This chapter briefly discusses the permission-based verification and parallelization approaches that are closely related to Java e.g., Plural, Pulse, Plaid and Æminium and the way they exploit the permission-based specifications. Finally, this chapter revisits the motivation example given in Listing 1.1 to elaborate on the efficacy and expressiveness of the permission-based specifications in enabling concurrency for sequential programs.

2.1 Access Permissions: An overview

Access Permission is a novel abstraction that encodes the information regarding how an object can be accessed through the current reference (method) as well as through possible other references (methods) in the system i.e., it combines the read, write and aliasing information of a referenced object.

There are three main categories of access permissions:

Fractional Permissions (Boyland, 2003). Fractional permission, fp , is a concrete mathematical value that defines shared ownership of objects in a concurrent setting where the share of each reference is between 0 and 1. The value 0 represents the absence of permission, whereas 1 represents `unique` (exclusive read and write) permission and any value greater than zero shows the read-only access for a shared object. Fractional permissions can be used to split whole permission to a number of fractions and distribute these fractions among multiple references at the same time. The fractional permissions can be split indefinitely. The splitting function for a `full` permission, say `fp`, when divided between two references, say fp_1 and fp_2, can be written as $fp_1 + fp_1 = fp$ with each reference having a share (fraction) of `s` in the range (0, 1).

Counting Permissions (Bornat et al., 2005). Counting permission is a special fractional permission where `s` is an integer value between 0 and a maximum `Integer` constant value, where zero represents the absence of permission and the maximum value represents `full` permission on the referenced object `o`. The read-only access on the referenced object is

represented by a non-zero integer value such that $0 < fp <= max$.

Symbolic permissions (Bierhoff and Aldrich, 2007). Symbolic permission, simply called access permission, is an extension of Boyland's permission sharing model but instead of using the fractional values to represent and split permissions among multiple references, the symbolic permission represents and tracks permission flow through the system using symbolic values such as `unique` or `immutable`.

There are five types of symbolic permissions that can be assigned to a reference `x`, for a referenced object `o`, in the presence of its alias `y`.

`unique(x)`: This permission provides reference `x` an exclusive read-write access on the referenced object `o` at any given time. No other reference (e.g. `y`) to the same object can co-exist while `x` has unique permission on it.

`full(x)`: This permission grants reference `x` with both read and write access to a referenced object `o`, and at the same time, the object `o` may also be read, but not written, by other references such as `y`.

`share(x)`: This permission is the same as the `full` permission, except that, in this case, other references such as `y` can also write on the referenced object `o`.

`pure(x)`: This permission gives reference `x` read but not write access on a referenced object `o`. Moreover, other references such as `y` may have both read and write access on the same object.

`immutable(x)`: This permission grants the non-modifying access on the referenced object `o` to both the current `x` and any other reference such as `y`.

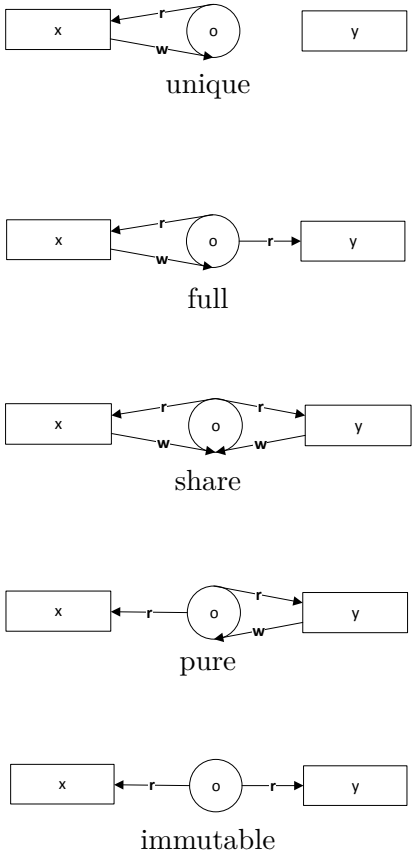

Figure 2.1: Five types of symbolic permissions

Table 2.1 summarizes how access permissions for a referenced object `o` can be assigned to this reference (`x`) and other reference (`y`) at the same time.

To generate five types of symbolic permissions from the source code of a Java program, the permission approach generates a permission-based graph model as shown in Figure 2.1, to represent the object's accesses and their aliasing information at the method level. In the generated model, `x` corresponds to the current method accessing a referenced object `o` and `y` corresponds to other methods ("the rest of the world") accessing the same object except the current method. The details of the permission inference and the graph construction mechanism are further explained in Chapter 3.

Table 2.1: Co-existing access permissions.

This Reference (x)	Access Rights	Other Reference (y)
`unique`	read/write	none
`full`	read/write	`pure`
`share`	read/write	`share`, `pure`
`pure`	read	`full`, `pure`, `immutable`
`immutable`	read	`immutable`, `pure`

2.1.1 Access Permission Splitting and Joining Rules

Linear logic (Girard, 1987) traditionally treats access permissions as resources that cannot be duplicated (discarded). Once a method consumes its permissions they are no longer available to other methods until this method returns the same permissions again. Access permission contracts in Linear Logic are specified using Linear Logic implication connective (⊸). The connective (⊸) operator is used to specify a method's pre- and post-conditions. As indicated by `P` ⊸ `Q`, permissions in the pre-conditions `P` are consumed before a method runs, and it produces `Q` as post-conditions when the method completes its execution.

Influenced by Boyland's work in (Boyland, 2003), Plural presented the fractional analysis of access permissions, hence, they can be split into one or more relaxed permissions i.e., fractions of the original permission using fractional values in the range (0,1) and then merged back into more restrictive or the original permission. The idea behind representing permissions as fractions is to explain when `write` permissions conflict with other permissions. A `write` permission requires to have the whole fraction (value 1) of the object permission to update state of the underlying object, so two aliases (references) to an object cannot simultaneously modify a memory location or read and write to it. However, two aliases of an object can coexist both with the read permissions. Access permission splitting and joining rules are given in Table 2.2. With fractional permission analysis, fractions keep tracks of the way the permissions were split and joined back. This information can be used to verify program behavior, based on specific criteria and parallelize execution of the program by tracking the permission flow through the system.

In Table 2, let x represent current reference, `o` represent the referenced object and k represent the fraction of permission assigned to a particular reference, where at least one of x_1 and x_2 is x, and $k_1 + k_2 = k$. The operator multiplicative conjunction (A ⊗ B) denotes simultaneous occurrence of permissions by multiple references, say x_1, x_2, on the same referenced object `o`. The symbol ⇐⇒ represents the two way operation of splitting and joining permissions. For example, a `unique` access permission (Rule-I) having `k` fractions can be divided into k_1 fraction of `full` and k_2 fraction of `pure` permission and then joined back accordingly. Likewise, a `unique` access permission (Rule III) can be split into two `share` permissions but cannot be split into a `share` and `immutable` permission as `immutable` cannot co-exist with the `share` permission. Linearity of resources forces the `unique` permission to be replaced by two

Table 2.2: Access permissions splitting and joining rules.

Splitting and joining Rules	**Rule #**
unique(x;o;k) $\Longleftrightarrow$ full(x1;o;k1) $\bigotimes$ pure(x2;o;k2)	Rule I
unique(x;o;k) $\Longleftrightarrow$ immutable(x1;o;k1) $\bigotimes$ immutable(x2;o;k2)	Rule II
unique(x;o;k) $\Longleftrightarrow$ share(x1;o;k1)$\bigotimes$ share(x2;o;k2)	Rule III
immutable(x;o;k) $\Longleftrightarrow$ pure(x1;o;k1)$\bigotimes$ immutable(x2;o;k2)	Rule IV
immutable(x;o;k) $\Longleftrightarrow$ immutable(x1;o;k1) $\bigotimes$ immutable(x2;o;k2)	Rule V
unique(x;o;k) $\Longleftrightarrow$ immutable(x1;o;k1)$\bigotimes$ immutable(x2;o;k2)	Rule VI
full(x;o;k) $\Longleftrightarrow$ share(x1;o;k1) $\bigotimes$ pure(x2;o;k2)	Rule VII
share(x;o;k) $\Longleftrightarrow$ full(x1;o;k1) $\bigotimes$ pure(x2;o;k2)	Rule VIII
share(x;o;k) $\Longleftrightarrow$ share(x1;o;k1) $\bigotimes$ pure(x2;o;k2)	Rule IX
share(x;o;k) $\Longleftrightarrow$ share(x1;o;k1) $\bigotimes$ share(x2;o;k2)	Rule X
full(x;o;k) $\Longleftrightarrow$ full(x1;o;k1) $\bigotimes$ pure(x2;o;k2)	Rule XI

`share` permissions which can be further split according to splitting rules and then joined back.

2.1.2 Access Permissions in the spirit of Design by Contract Principle

Contract-based specifications provide a good way to distinguish and convey to the readers, the intended behavior of a software application in an abstract way. The use of abstract representations hides the implementation details from the caller of the method thus maintaining the information hiding principle. A contract is an abstraction to specify a method's behavior. In the Design by Contract principle (Meyer, 1988), contracts are obligations and rights of the client and the implementing class itself. Contracts are specified using the `requires` and the `ensures` clauses that represent a method's pre and post-conditions respectively (Meyer, 1992; Leavens et al., 2006).

In the spirit of the Design by Contract principle, permission-based specifications at method level represent *contracts* where permission-based obligations are defined as pre-conditions `P` that client of a class must guarantee before calling methods of the class, and permission-based rights represent post-conditions `Q` that must hold for both the client and the implementing class after executing the specified method. The idea of specifying pre- and post-conditions as contracts dates back to Hoare's work (Hoare, 1969) on formal verification of software applications and has recently been applied to permission-based verification and parallelization approaches (Cataño et al., 2014; Stork et al., 2014; Huisman and Mostowski, 2015; Müller et al., 2017).

2.2 Permission-based Program Verification in Plural and Pulse

To the best of our knowledge, there are only two research tools that are directly related to Java-based access permissions, namely Plural[1] (Bierhoff and Aldrich, 2008) and Pulse[2] (Siminiceanu et al., 2012).

It is worth mentioning here that the permission inference approach, presented in this thesis, integrates the Pulse tool to perform the correctness and concurrency analysis of the inferred specifications. The Pulse tool accepts a Java program annotated with the Plural specifications i.e, access permission contracts and typestate information.

The goal of this section is to describe the syntax and semantics of access permissions in the Plural specification language and explain the permission checking mechanism in the Pulse tool. This is because the permission inference framework borrows the Plural's syntax to generate the Pulse translated version of the input Java program. It further extends the existing concurrency analysis in the Pulse tool to check the potential for concurrency, in a sequential Java program, based on five types of symbolic permissions. Therefore, it is non-trivial for the readers of the automatic permission checking approach, in our proposed framework, to understand the Plural specifications and their concurrency analysis in Pulse.

2.2.1 Plural

Plural (Permissions Let Us Reason about Aliases) (Bierhoff and Aldrich, 2007) is a formal specification language and a tool originally developed to ensure protocol compliance in typestate-based sequential programs such as Java APIs. The aim was twofold, firstly to verify protocol conformance with actual program implementation in the presence of aliasing, and secondly to check whether a client of the program obeys the specified protocol.

In Plural, programmers explicitly specify their design intents using permission-based typestate contracts at the method level where access permissions represent the read and write behavior of a method on the referenced objects and their aliasing information. The typestates (Strom and Yemini, 1986) describe the set of valid object's states a method can be called on. Plural performs intra-procedural static analysis, called DFA (Diagram Flow Analysis) of the annotated program to identify and track the specified pre- and post-permissions across all the method calls for every program variable (parameter, receiver object, and local variable). It checks each method separately and ensures that all the declared pre-conditions are met at the call sites and issues warnings for protocol violations in the program.

The technique is implemented in a tool (Bierhoff and Aldrich, 2008), a permission-based automated protocol checking and conformance tool implemented as a Java Eclipse plugin. It supports five types of access permissions such as unique, immutable, full, pure and share as parts of method specifications.

In a Plural program, the annotation `@Perm` is used to specify a permission contract,

[1] Pluralism: Modular Object Protocol Checking for Java, https://code.google.com/archive/p/pluralism/

[2] Pulse: A Model-checking tool to verify typestates and access permissions specifications of Java programs, http://aeminium.dei.uc.pt/index.php/ToolsAndDownloads

following the Design by Contract principle, where pre- and post-conditions are defined using `requires` and `ensures` clauses respectively. A typestate in Plural is declared using `@State` clause and multiple typestates are declared inside `@ClassStates` declaration. Typestate '`Alive`' is a default global state an object can be in. The precondition (`requires` clause) in a method contract specifies the type of access permissions (`AP`), a method requires on a referenced object `this` using notation `AP(this)` and the typestate (`s`), a referenced object should be in before the method starts its execution. The permission (`AP`) on a parameter is represented using notation `AP(#i)` where `i` is an integer that maps the position of a parameter in a method declaration as $0 \rightarrow N-1$ where `N` is the number of parameters in a method signature. The post-condition (`ensures` clause) in a method contract specifies the permissions (`AP`$'$) that the method should generate on the referenced object to return it back to the caller method and the typestate (`s`$'$) an object should hold when the method exits. The symbol $*$ shows the multiplicity (one or more) of the referenced objects with permission annotations. The notation `ENDOFCLASS` is used to distinguish between multiple classes in the Pulse input program.

Listing 2.1 shows a sample class `Task` taken from a case study MTTS with Plural annotations. MTTS is an industrial application in Java, developed by Novabase company[3] and has been extensively been used in financial sectors to parallelise computational tasks over multiple servers. The case study was verified using the Pulse tool (Cataño et al., 2014) based on permission-based typestate contracts.

Listing 2.1: A sample code snippet for Plural annotated Java program in Pulse (Cataño et al., 2014)

```
1 import edu.cmu.cs.plural.annot.*;
2 @ClassStates({
3 @State (name= "Created"),
4 @State (name= "Ready"),
5 @State (name= "Filled"),
6 @State (name= "Completed"),
7 @State(name = "Alive")})
8 public class Task {
9  private TaskData data;
10  @Perm(ensures= "unique(this) in Created")
11  public Task(){ }
12  @Perm(requires= "full(this) in Created * pure(#0) in Filled",
13        ensures= "full(this) in Ready * pure(#0) in Filled")
14  public void setData(TaskData d){ ... }
15  @Perm(requires= "pure(this) in Ready",
16        ensures= "pure(this) in Ready")
17  public TaskData getData(){ ... }
18  @Perm(requires= "full(this) in Ready",
19        ensures= "full(this) in Completed")
20  public void execute(){ ... }
21  @Perm(requires= "pure(this) in Alive",
22        ensures= "pure(this) in Alive")
23  public int getStatus ( ) { ... }
24 }
25 ENDOFCLASS
```

In MTTS program, the class `Task` captures all the information about a generic task in a data structure `(data)` in Line 9. The specifications (Line 10) on the constructor method `Task()` show that it does not require any permission on the receiver object `this` as there is no '`requires`' clause, but it ensures, using the `ensures` clause, that a new task must be created with `unique` permission, represented by `unique(this)` annotation in the `Created` state.

[3] http://www.novabase.pt/pt

Likewise, the precondition of method `setData()` in Line 12 specifies that the method requires full permission, in the `Created` state, on the receiver object `this` before updating the value of its `data` object. It further specifies that the method needs `pure` permission on parameter 'd' represented by `pure(#0)`, in the `Filled` state. The post-condition of method `setData()` (Line 13) specifies that the method should generate the same permissions (`full(this) & pure(#0)`) on the referenced objects in their respective typestates, to return the consumed permissions back to the caller of the method. The rest of the methods in Listing 2.1 would be discussed, in the next section, to elaborate the correctness analysis of the Plural specifications by Pulse.

2.2.2 Pulse

Pulse (Siminiceanu et al., 2012) is a formal verification approach and a tool, developed at the University of Madeira, that verifies the correctness of the Plural specifications defined at the method level. It accepts a Java program annotated with the Plural specifications. Unlike Plural, it verifies the correctness of the input specifications itself, in isolation to the program implementation. The goal was to help programmers write semantically correct specifications to help verify the correctness of the program based on access permissions.

Figure 4.3 shows the workflow to verify Plural specifications through the Pulse tool.

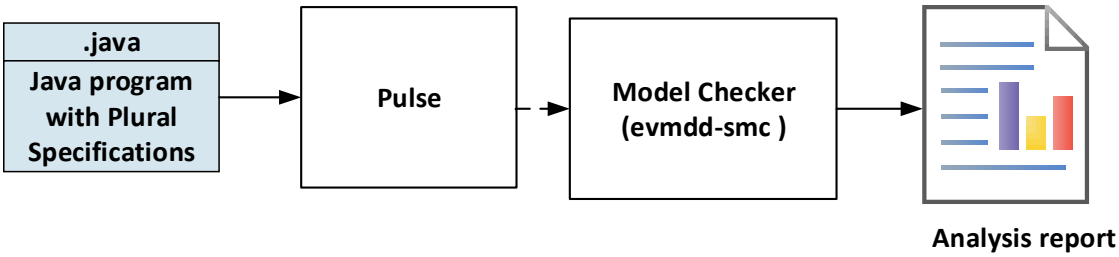

Figure 2.2: A high-level workflow of the Pulse tool.

To perform a comprehensive analysis of the input specifications, Pulse translates the Plural specifications into a semantically equivalent abstract state-machine model. The model captures the dynamic behavior of a program as a sequence of method calls obeying the access permission semantics, and the typestate information associated with the referenced objects. Pulse employs the evmdd-smc symbolic model-checker (Roux and Siminiceanu, 2010) to verify the machine models.

The model checker ensures that the input specifications satisfy a set of core integrity properties specified as Computation Tree Logic (CTL)(Huth and Ryan, 2000) formulae. CTL models time in a tree structure commonly called a computational tree. The model checker identifies the a) absence of sink states (deadlocks), b) method's satisfiability, c) access permissions correctness, d) possible concurrency among methods, and e) possible transitions among typestates, based on the input specifications, following the pre-defined CTL formulae given below.

The following sections briefly explain the step-by-step access permission encoding and model checking mechanism in Pulse and elaborate it with the Plural annotated Java program

given in Listing 2.1. The details of the approach can be found in (Siminiceanu et al., 2012; Cataño et al., 2014).

Generating Abstract State-Machine Model for the Plural Specifications

Pulse generates the abstract state machine model of the Plural specifications to capture the dynamic behavior of the object references in the input specifications. The model is then checked against the method's specifications (pre- and post-permissions, typestates and typestate invariants, if any), irrespective of their implementation.

A. Abstract Model of the Plural Specifications

A Plural specification comprises a finite set of class declarations $\mathbf{C} = \{\mathbf{C_1}, \ldots, \mathbf{Cc}\}$ such that:

- Every class C_i, for $1 \leq i \leq c$, contains a set of typestate declaration ts_i and set of methods M as shown below.
- For every class $C_i \in C$, the approach identifies each object o_i from the set of objects $\mathbf{O} = \{\mathbf{o_1}, \ldots, \mathbf{on}\}$ in the specifications (including method's parameters and class fields).
- All the objects $o_i \in O$ are mapped to their class declarations using an implicit mapping of the form $\mathbf{class\_of} : \mathbf{O} \rightarrow \mathbf{C}$.
- Further, to analyze a truly concurrent behavior of the system, for each object o_i , $1 \leq i \leq n$, the approach creates a number of instances of references to that object as $\mathbf{R_i} = \{\mathbf{r_i^0}, \mathbf{r_i^1}, \mathbf{r_i^j}, .., \mathbf{r_i^K}\}$, where K is a user-defined integer (parameter) that shows the number of distinct aliases for each reference. For $K = 0$, there would be no concurrency in the model. However, the value of K should strictly be a positive value ($K > 0$) to allow concurrency in the generate model.

  The building block of the generated model is the state-machine of an object reference r_i^j, where $\mathbf{h} = \mathbf{class\_of(i)}$, which includes the following main components.

  - An abstract program counter (pc_i^j), is the set of methods (constructors) defined for object o_i,

$$pc_i^j \in PC_i = \{\mathbf{exe}, \mathbf{done}\} \times (\{\perp\}), M_h = \{M_h^1, M_h^2, \ldots, M_h^{m_h}\} \tag{2.1}$$

  - The access permissions (ap_i^j) associated with the reference r_i^j are defined as

$$ap_i^j \in AP = \{\perp, \mathbf{Unique}, \mathbf{Full}, \mathbf{Share}, \mathbf{Pure}, \mathbf{Immutable}\}. \tag{2.2}$$

  - A typestate ts_i associated with each object 0_i is defined as

$$ts_i \in \tau s_h = \perp \cup \{t_h^1, t_h^2, \ldots, t_h^{lh}\} \tag{2.3}$$

  where l_i is the number of typestates for class C_i, for $1 \leq i \leq c$ and the symbol $\perp$ represents the undefined values for typestates, access permissions and methods in the input specifications. The symbols `exe` and `done` are explained below.

State Transition Rules

The approach defines state transition rules to allow a non-deterministic transition between the two pre-defined states at the method level. A *done* local state specifies a local state for method m with a program counter $\mathbf{pc_i^j} = (\mathbf{done}, \mathbf{m})$ while an *exe* local state is a local state with the program counter represented as $\mathbf{pc_i^j} = (\mathbf{exe}, \mathbf{m})$.

- The model allows a transition from a *done* **local state** to any other *exe* **local state** if the specifications respect a **pre-defined guard formula** (enabling conditions). The guard formula with all the defined conditions (conjuncts) should hold for a transition to be enabled for method m. It captures:
  - the required or the compatible access permissions, following the access permission splitting and joining rules given in Table 2.2, from the input specifications, and
  - the required typestate to make the transition to the `exe` local state.
- **From each** *exe* **local state** i.e., (exe, m), a reference can only transition to its matching `done` local state i.e, $(done, m)$, capturing the completion of the method call for method m.

Additionally, each transition is guarded by a post-condition and **an update formula**, associated with the method specifications, that reflects the expected change in the access permissions and associated typestate, after the execution of the method.

B. Access Permissions Translation Mechanism

To convert the Plural specifications into an abstract state machine model acceptable by the `evmdd` model checker, the translation mechanism builds two components of a finite state machine: the set of global states S and the transition relation between states i.e., $R \subseteq S \times S$. The potential state space is simply a cross product of the local state spaces of n objects which includes the typestate information (common to all of the references of an object), the program counter and the access permissions for each of the $K+1$ references.

$$S = \prod_{i=1}^{n} \tau s_i \times \prod_{j=0}^{K} (PC_i \times AP)$$

There are two local transitions, for each reference r_i^j, corresponding to the `start` and `end` of a method m. The notation `(from-states, to-state)` represents the pairs of states in the transition relation where *unprimed* variables refer to `from-state` and the *primed* variables refer to the `to-state` part of the relation.

The input for a transition relation is the calling reference r_i^j itself, the method m, two triples of the form **Triple** = (**Reference**, **Typestate**, **AccessPermission**) and the global states i.e., an array of system's local states. The two triples i.e., $(r_{i_0}^{j_0},\ ts_{i_0}^{k_0},\ ap_0)$, $(r_{i_1}^{j_1},\ ts_{i_1}^{k_1},\ ap_1)$, encodes the `requires` (indexed i_0) and the `ensures` (indexed i_1) clauses that basically represents the `required` and `ensured` information (typestate, reference, and access permission) in the method's specifications. The approach tracks and verifies the state transition between the starting and ending of the method m, for each reference r_i^j, by following the **guard formula**. It further updates the changes in the values of the global

states, if required, by following the **update formula**.

- For example, **to start a method** m for a reference r_i^j, the approach checks if the reference r_i^j is not already executing method m, the global typestate associated with object i_0 is the required typestate $ts_{i_0}^{k_0}$, and the pre-permission of object i_0 is compatible with the current permission ap_0, and the post-permission of object i_1 is compatible with ap_1. If all the conjuncts of the guard formula hold for this transition, the approach updates the global state and the program counter of reference r_i^j to be "executing the method" i.e., `(exe, m)`.
- **Similarly, to end a method** m by a reference r_i^j, the approach checks if the method m is the one currently executed if this condition holds, the approach updates the typestate of reference $r_{i_1}^{j_1}$ to its `ensured` typestate, its pre-permissions to post-permission ap_1, and the status of the program counter, for the calling reference r_i^j, to be "done" i.e., `(done, m)`.

Access Permission Compatibility and Transformation

To check the permission compatibility and transformation of permissions in the two triples, the approach employs the Boyland's fractional permission model (Boyland, 2003) to explain when the "write" permissions conflict with other permissions. A write (`unique`) permission, represented by value 1, consumes the whole fraction of permission to write on a referenced object, so two references cannot write on the same object or read/write on it at the same time. However, two read permissions such as `immutable`, with a fraction value between 0 and 1, can coexist on the same object that allows other references to access the same object. The approach follows the access permission splitting and joining rules in Table 2.2 where the whole (fraction) permission is the sum of its fractions say $k = k_1 + k_2$, across the references $r_{i_0}^{j_0}$ and $r_{i_1}^{j_1}$.

The approach checks the permission compatibility, following the permission co-existence rules given in Table 2.1, to decide if a permission ap_0 can be downgraded or upgraded to another permissions ap_1 and uses this information to transform the permissions from ap_0 to ap_1 when and where required. For example, the `full` permission can be downgraded to `immutable` permission, if it gives up its write access that can be used by the `other` references. However, the `share` and `immutable` permission are not compatible with each other so they cannot be transformed into each other.

Discrete State Semantics for Fractional Access Permissions

The approach generates the discrete state semantics for the input specifications. It considers the access permissions as globally available resources (tokens), stored in a central bank **B**, that can be divided among multiple references of the same object. Using the bank analogy, a reference can take (borrow) a fraction of a token (permission) or all tokens, depending on its pre-condition and it returns the consumed tokens to the bank once finished its processing. The approach ensures that the total number of tokens for each object remains preserved, with the unused fractions always in the bank, thereby maintaining a global invariant property that no resources are created or lost for the individual objects.

The underlying approach defines the access permissions of a reference r_i^j as a pair of

fractions, $(fr_i^j,\ \ fw_i^j)$ in the range [0, 1], representing the read and write access to the referenced object o_i. The possible combinations of the fraction (permission) values, following the permission semantics in Section 2.1, for the current (`this`) and other (`O`) reference is defined as shown in Figure 2.3.

this	Semantic	Bank	O
$fr_i^j = 0 \wedge fw_i^j = 0$	**null**	$fr_i^B \geq 0 \wedge fw_i^B \geq 0$	any
$fr_i^j = 0 \wedge fw_i^j > 0$	no meaning	-	-
$0 < fr_i^j < 1 \wedge fw_i^j = 0$	**Immutable**	$fw_i^B = 1$	$\sum_{l \neq j} fw_i^l = 0$
$0 < fr_i^j < 1 \wedge fw_i^j = 0$	**Pure**	$fw_i^B < 1$	$\forall l \neq j : fw_i^l \geq 0$
$0 < fr_i^j, fw_i^j < 1$	**Share**	$0 \leq fr_i^B < 1 \wedge 0 \leq fw_i^B < 1$	$\forall l \neq j : 0 \leq fr_i^l, fw_i^l < 1$
$0 < fr_i^j < 1 \wedge fw_i^j = 1$	**Full**	$0 \leq fr_i^B < 1 \wedge fw_i^B = 0$	$\forall l \neq j : fw_i^l = 0$
$fr_i^j = 1 \wedge fw_i^j = 0$	**Immutable**	$fr_i^B = 0 \wedge fw_i^B = 1$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$
$fr_i^j = 1 \wedge fw_i^j = 0$	**Immutable**	$fr_i^B = 0 \wedge fw_i^B < 1$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l > 0$
$fr_i^j = 1 \wedge 0 < fw_i^j < 1$	no meaning	-	-
$fr_i^j = 1 \wedge fw_i^j = 1$	**Unique**	$fr_i^B = 0 \wedge fw_i^B = 0$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$

Figure 2.3: A fractional permission model in Pulse (Cataño et al., 2014).

The implementation generates a fully discrete model of the specifications by mapping the permission (fraction) values in the range $[0,\ \ K+1]$, As discussed previously, K represents the maximum number of independent aliases for the object references below:

- The fractions from the continuous interval [0, 1] are mapped to a set {0,1, . . . ,K + 1} using the abstraction in Equation 2.4.

$$\mathrm{N} : [0,\ \ 1] \rightarrow \ \ \{0, 1, \ldots, K{+}1\}, \quad N(f) = \begin{cases} 0, & \text{if } \ f = 0 \\ x \in \{1, \ldots, K\}, & \text{if } f < 0 < 1 \\ K{+}1, & \text{if } \ f = 1 \end{cases} \tag{2.4}$$

  where $f : f = 0$ represent no permission, the value $< f < 1$ represent the partial/share permissions, and $f = 1$ shows the exclusive rights owned by reference r_i^j to the object 0_i.

- The required number of read and write tokens, Nr and Nw, are mapped, to represent the minimum amount of resources required for the next operation, as follows:

$$N_r : AP \ \ \rightarrow \ \ \{0,1, \ldots, K{+}1\}, \quad N_r(a) = \begin{cases} 0, & \text{if } a = \perp \\ 1, & \text{if } \ a \in \{\text{Full, Pure, Immutable, Share}\} \\ K{+}1, & \text{if } \ a = \text{Unique} \end{cases} \tag{2.5}$$

$$N_w : AP \quad \rightarrow \quad \{0,1,\ldots,K+1\}, \quad N_w(a) = \begin{cases} 0, & \text{if a} \in \{\bot,\ \text{Pure},\ \text{Immutable}\} \\ 1, & \text{if a} = \text{Share} \\ K+1, & \text{if a} \in \{\text{Unique},\ \text{Full}\} \end{cases} \tag{2.6}$$

- Further, the number of read and write tokens for each reference r_i^j are represented as `tkr` and `tkw` in the generated model, as follows. The unprimed variables refer to values before the transition (`from-state`), while the primed variables represent the values after the transition (`to-state`).

$$tkr_i^B + tkr_i^j \geq 1 \quad \rightarrow \quad tkr_{i\prime}^j = 1 \wedge tkr_{i\prime}^B = tkr_i^B + (tkr_i^j - 1) \tag{2.7}$$

$$tkr_i^B + tkr_i^j = 0 \wedge \exists h \neq j : pc_i^h = (done,.) \quad \wedge \quad tkr_i^h \geq 1, \rightarrow \quad tkr_{i\prime}^j = 1 \wedge tkr_{i\prime}^h = tkr_{i\prime}^h - 1 \tag{2.8}$$

  For example, if a method requires non-exclusive access rights, say `pure` permission on the current reference r_i^j to start its execution, the guard checks whether the current reference already has one read token necessary for starting the method, (resulting in the transition given in Equation 2.7), or if it needs to "borrow" it from some other reference r_i^h, which has already completed its execution with $(done, m)$ state, as is the case in the Equation 2.8.

D. Model Generation

The approach then encodes the generated discrete state semantics of the input specifications to an abstract state machine model, in the input language of `evmdd-smc` model checker. For which it first defines the abstract domain of the model and generates four sections a) variable declarations, b) variable initializations, c) the transition relation, and d) a set of desired properties.

a). Abstract Domain: The abstract domain is generated by declaring all the variables in the input specifications as discrete (integer interval type) in the model as follows.

- The **method's identifiers** domain (Equation 2.1) is mapped to $[\mathbf{0}, \mathbf{m_i}]$.
- The domain for the **access permission types** (Equation 2.2) is mapped to $[\mathbf{0}, \mathbf{5}]$, ranging from `none` to the most restrictive permission i.e, `unique`
- The **typestate domain** (Equation 2.3) is mapped on the interval $[\mathbf{0}, \mathbf{h_i}]$. The value 1 is reserved to represent the `Alive` typestate, that is a root typestate. A transition is allowed from any typestate to `Alive` typestate in the generated model.
- The **read/write token domain** for the variables is mapped to $[\mathbf{0}, \mathbf{K+1}]$ and,
- The **domain** for the type of **local states** `(exe, done)` is mapped to $[\mathbf{0}, \mathbf{1}]$ interval.

b). Variable Declaration: For each object o_i, the approach declares two types of variables in the model:

- The **first category** is for the actual object o_i which includes three variables i.e., $state_i$ of

the domain ts_i, the read and write tokens (tkr_i^B and tkw_i^B) in the bank of type [0,K+1].

- The **second category** is for the references r_i^j to the object o_i. It defines five variables for each of the (K+1) references to the object i.e., program counter pc_i^j of type [0, 1], the $method_i^j$ of type [0, m_i], access permissions ap_i^j of type [0, 5], and read/write tokens tkr_i^j and tkw_i^j of type [0, K + 1].
  In total, the model contains $c*(3+5*(K+1))$ variables.

c). Variable Initialization: The approach initializes the model by defining initial values for each object o_i and each reference r_i^j to the object in the following ways:

- **For each object** o_i, initially, the typestate is defined in an unknown state as $state_i = \perp(0)$, and all the read and write tokens are stored in the bank $tkr_i^B = K + 1$ and $tkw_i^B = K + 1$
- **for each reference** r_i^j: for $0 \leq j \leq K$, the $pc_i^j = done(1)$ (i.e., reference is not executing something else), $ap_i^j = \perp(0)$ (i.e., none permission), $method_i^j = \perp(0)$ and the read and write tokens are set to zero as $tkr_i^j = 0$ and $tkw_i^j = 0$.

Figure 2.4 shows the variable declaration and initialization sections for reference r_0^0, for the `TaskData` class, given in Listing 2.1, in the generated model. The value of K, in this case, is set as $K = 4$.

```
Declarations
/*Task-->*/
    state_Task_0        [0, 4]
    tkrB_Task_0         [0, 5]
    tkwB_Task_0         [0, 5]

    pc_Task_0_0         [0, 1]
    method_Task_0_0     [0, 5]
    ap_Task_0_0         [0, 5]
    tkr_Task_0_0        [0, 5]
    tkw_Task_0_0        [0, 5]
/*Task-->*/
```

(a) Variable declarations.

```
Initial states
/*Task-->*/

    state_Task_0        = 0
    tkrB_Task_0         = 5
    tkwB_Task_0         = 5

    pc_Task_0_0         = 1
    method_Task_0_0     = 0
    ap_Task_0_0         = 0
    tkr_Task_0_0        = 0
    tkw_Task_0_0        = 0

/*<--Task*/
```

(b) variable initializations.

Figure 2.4: The evmdd-smc model for variable declarations and initialization of the TaskData class

c). Defining Transition Relations: The state transition rules are already explained in the previous sections. The unprimed variables refer to values before the transition (`from-state`), while the primed variables represent the values after the transition (`to-state`). There are four categories of transitions in the generated model described below:

Transitions:

1. A transition where the reference r_i^j starts a constructor.
   - The **guard expression** ensures that the object has not yet created $\bigwedge_{j=0}^{K} ap_i^j = \perp$.
   - The **update expression** is $pc_{i\prime}^j = exe, \wedge\ method_{i\prime}^j = constructor \wedge ap_{i\prime}^j = Unique \wedge tkr_{i\prime}^B = 0 \wedge tkw_{i\prime}^B = 0 \wedge tkr_{i\prime}^j = K+1 \wedge tkw_{i\prime}^j = K+1$

2. A transition where the reference r_i^j starts a non-constructor method m_i^k.
   - The **guard expression** ensures that r_i^j exists and $ap_i^j \neq \bot \wedge pc_i^j = (done,.) \wedge state_i = t_i^h \wedge tkr_i^B \geq tkr_i^k \wedge tkw_i^B \geq tkw_i^k$
   - The **update expression** is $pc_{i\prime}^j = (exe, .) \wedge method_{i\prime}^j = k \wedge ap_{i\prime}^j = ap \wedge tkr_{i\prime}^B = tkr_i^B - tkr_i^k \wedge tkw_{i\prime}^B = tkw_i^B - tkw_i^k \wedge tkr_{i\prime}^j = tkr_i^j + tkr_i^k \wedge tkw_{i\prime}^j = tkw_i^j + tkr_i^k$
3. A transition where the reference r_i^j ends a non-constructor method m_i^k.
   - The **guard expression** ensures that r_i^j exists and it is currently executing the method m_i^k. The guard is $pc_i^j = (exe,;.) \wedge method_{i\prime}^j = k$
   - The **update expression** is $pc_{i\prime}^j = (done, .) \wedge tkr_{i\prime}^B = tkr_i^B + tkr_i^k \wedge tkw_{i\prime}^B = tkw_i^B + tkw_i^k \wedge tkr_{i\prime}^j = tkr_i^j - tkr_i^k \wedge tkw_{i\prime}^j = tkw_i^j - tkr_i^k$.
4. A transition where the reference r_i^j is a newly created alias.
   - The **guard expression** ensures that r_i^j is not previously created and $state_i \neq \bot$ $land\ ap_i^j = \bot \wedge tkr_i^B \geq 1$
   - The **update expression** is $pc_{i\prime}^j = (done, .)\ , \wedge method_{i\prime}^j = \bot, \wedge ap_{i\prime}^j = Pure$

   Figure 2.4 shows the `evmdd-smc` code for the constructor method `TaskData()`, in Listing 2.1, with its `start` and `end` transition and their `guard` and `update` formulae for reference r_0^0.

```
Transitions
/* for K = 0-->*/

/*Task*/
    start_Task_Task_0_0:
    ap_Task_0_0 = 0  -> /* constructor guard * /

    /* update formula */
    pc_Task_0_0' = 0  /\
    method_Task_0_0' = 1  /* Constructor*/  /\
    ap_Task_0_0' = 1  /* unique */  /\
    tkrB_Task_0' = 0 /\ tkwB_Task_0' = 0 /\
    tkr_Task_0_0' = 5    /\ tkw_Task_0_0' = 5

    end_Task_Task_0_0:
      pc_Task_0_0 = 0 /\ method_Task_0_0 = 1 /* constructor , guard */
      ->
      /* update formula */
      pc_Task_0_0' = 1 /\
      state_Task_0' = 1 /* alive */ /\
      tkrB_Task_0' = 5 /\ tkwB_Task_0' = 5  /\
      tkr_Task_0_0' = 0 /\ tkw_Task_0_0' = 0

/*Task*/
```

Figure 2.5: The evmdd-smc model for the Start and End Transitions of the Constructor method in TaskData class

d). Properties: The model checker then verifies the correctness of input specifications (access permissions and typestate information) for which the approach defines core set of guarantees, as generic formulae in CTL, given in the next section.

CTL Properties

This section explains the CTL formulae and elaborates the Pulse permission checking mechanism through them, using the Plural annotated program given in Listing 2.1. CTL formulae combine the temporal operators and path quantifiers from the Linear Temporal Logic (LTL)(Pnueli, 1977), a formalism that describes a system as a sequence of state transitions. It defines time in terms of temporal operators such as `eventually` and `never` to describe the behavior of reactive systems. The term `eventually` specifies the desirable behavior of a system and `never` is used to specify that something undesirable would not happen ever (Grumberg and Veith, 2008).

Correctness of Access Permissions: The model checking approach enforces that access permissions do not violate their intended semantics by defining a discrete state semantics model for them, as explained in Section 2.2.2. In the CTL formula, the semantics are expressed as:

$$\forall\, 1 \leq i \leq c, \forall\, m \in M_i, 0 \leq j_1 \neq j_2 \leq K\colon$$

$$unique(m)\colon EX(pc_i^{j_1} = (m, exe) \wedge ap_i^{j_2} = \bot) \tag{2.9}$$

$$not\_full(m)\colon EX(pc_i^{j_1} = (m, exe) \wedge pc_i^{j_2} = (., exe) \wedge tkw_i^{j_2} > 0) \tag{2.10}$$

For example, if a method requires `unique` permission on a referenced object (`this`), following the discrete semantics for this permission, it requests all the tokens from the bank i.e., $fr_i^j = 1, fw_i^j = 1$, leaving no fraction for the other references to read or write on the same object i.e., $fr_i^B = 0, fw_i^B = 0$ and $fr_i^l = 0, fw_i^l = 0$, represented by $ap_i^{j_l} = \bot$ in Equation 2.9. In this case, the program counter for the current reference $r_i^{j_1}$ would be in "executing" (`m, exe`) state while the permission for other references would be `none`. Similarly, Equation 2.10 defines the CTL formulae in case of `pure` permission.

Methods Satisfiability Analysis: The predicate $satisfiability_i(m)$, represented as the CTL property below, is used to check whether the pre-conditions (access permissions or typestates) of a method m is met. Alternatively, this property checks if a method can transition to the `exe` local state state. The unsatisfiability of the `requires` clause may be because of the non-availability of the required access permissions or the typestates in the input specifications which shows, that the method will never be called by another object.

$$\forall\, 1 \leq i \leq c, \forall\, m \in M_i\colon satisfiability_i(m)\colon EX(pc_i^j = (m, exe)) \tag{2.11}$$

For example, the method satisfiability analysis of the MTTS program, given in Listing 2.1 by the Pulse tool reports methods `setData()` and `execute()` as unsatisfiable. This is because their pre-conditions are not met.

The constructor method produces an object with `unique` permission and in typestate `Created`. According to the access permission splitting and joining rules 2.2, `unique` permission can be split into `full` and `pure` permission. The `full` permission along with typestate `Created` can satisfy the first part of the `requires` clause for method `setData()`. However, the second part of the method cannot be satisfied, as no method transitions object (`d`) into the `Filled` state, hence, this method remains unreachable. The unsatisfiability of method `setData()` causes the unreachability of method `getData()` and `execute()`, due to the non-availability of the typestate (`Ready`) required by both the methods.

Concurrency Analysis: The predicate $concurrent(m_1, m_2)$, in the CTL formula given below, is used to check whether two methods can be run in parallel, meaning that two program counters for methods m_1 and m_2 exist with the local state value `exe`. An empty set of states satisfying the predicate $concurrent(m_1, m_2)$ shows that method m_1 and m_2 cannot be executed in parallel.

$$\forall\, 1 \leq i \leq c, 0 \leq j_1 \neq j_2 \leq K, \forall m_1 \neq m_2 \in M_i = \\ concurrent_i(m_1,m_2) \colon EX(pc_i^{j_1} = (m_1, exe) \wedge pc_i^{j_2} = (m_2,exe)) \tag{2.12}$$

Table 2.3 shows the pulse concurrency analysis for the MTTS program given in Listing 2.1. It reveals that the constructor method `Task()` cannot be parallelized with any other method as no other method can be in `exe` local state state at the time of object creation. Further, it shows that the methods requiring modifying access (`full`) on the same object cannot be executed in parallel. However, methods that require read `pure` access such as method `getData()` and `getStatus()`, on the same object can safely be executed with other methods.

Table 2.3: Method concurrency matrix in MTTS

	Task	setData	getData	execute	getStatus
Task	∦	∦	∦	∦	∦
setData	∦	∦	∥	∦	∥
getData	∦	∥	∥	∥	∥
getStatus	∦	∥	∥	∥	∥
execute	∦	∦	∥	∥	∥

Checking Sink states: The presence of states without the sink state may lead to a circular wait for the next resource and eventually the deadlock. The presence of unreachable states may arise due to a method's unsatisfiable pre-conditions or the improper use of access permissions. The sink states in the generated model are represented as CTL formula given below:

$$deadlock : \exists EX(true) \tag{2.13}$$

The permission inference approach, presented in this research, integrates the Pulse tool to perform the correctness, concurrency, and method satisfiability analysis of the access permissions. However, inferring (verifying) the typestates information is not an immediate objective of this research.

Chapter 3 presents the permission checking mechanism for the motivating example given in Listing 4.2, along with the extension made by our approach, in the Pulse permission checking mechanism, as a part of its concurrency analysis.

2.3 Permission-based Program Parallelization in Plaid and Æminium

This section briefly explains the two permission-based programming paradigms e.g., Plaid and Æminium that have been developed to parallelize execution of single-threaded programs based on access permissions.

2.3.1 Plaid

Plaid (Aldrich et al., 2011, 2012) is a new permission-based programming paradigm, with a new type and runtime system, developed to parallelize the execution of typestate-based sequential programs. The aim was to extend the typestate-oriented programming (Aldrich et al., 2009), that was originally designed to verify program behavior by tracking the state of a reference object at runtime, with the first class states and access permissions.

Every type in Plaid is represented as tuples having a type structure and associated permissions to express aliasing and mutability of the corresponding object's typestate. Plaid borrows its grammar and lexical structure from the Java Specification Language (JSL) but it shows significant differences from Java to incorporate permission-based specifications as part of the language. Everything in Plaid is an object, including the primitive types in Java, which are mapped to their corresponding wrapper objects in Plaid. It does not support Java modifiers. However, it provides interoperability with Java. Using Plaid, we can call Java methods by developing wrapper methods in a Plaid program. Unlike Java, it represents classes using keyword `state` and transitions between state are represented by a state transition symbol « which distinguishes the pre-state from the post-state.

Plaid supports three types of access permissions: unique and immutable for individual objects, and share permissions for the collection objects. The keyword 'none' is used to represent the absence of permissions or when no permissions are required on a referenced object. In the context of access permissions, the symbol « is used to specify permission contracts following the syntax `pre-condition « post-condition` with the signature of a method to show how a method changes the state of its arguments (receiver object). In the permission contract, the part after « symbol can be omitted if the method does not change the permissions associated with its referenced objects which means that the `pre-` and `post-permissions` are the same. For example, the permission contract for the parameter `amount` in Listing 2.2 in Line 4 can be written as `immutable Amount amount` without explicitly mentioning the `post-permission`.

Listing 2.2 shows a sample method `transfer()`, for a `BankAccount` class, with permission-based annotations in Plaid. The method `transfer()` handles `deposit()` and `withdraw()` transactions to transfer a given amount between two bank accounts. The method first withdraws the specified `amount` from the `sender` account in Line 6, and then deposits the same amount into the `receiver` account (Line 7). The pre-permission (before the « symbol) associated with the method's signature (Line 2) states that it requires `unique` permission on the `sender` and the `receiver` object representing the `exclusive` access on these objects. The post-permission (after the « symbol) specifies that the method will return the same permission back to the caller of the method when it completes its execution. The annotation `immutable` on parameter `amount`, in Line 4, shows that other methods (aliases) can access the `amount` variable but none of them can change it.

Listing 2.2: A sample method specification in the Plaid language (Stork et al., 2014).

```
1 state BankAccount{
2 method void transfer(unique Account sender << unique sender,
3                       unique Account receiver << unique receiver,
4                       immutable Amount amount) {
5  //sender : unique, receiver : unique, amount : immutable
6  withdraw(sender, amount);
7  deposit(receiver, amount);
8 }
9 }
```

2.3.2 Æminium

Having access permissions support in the Plaid infrastructure, Stork et al. (2014) proposed a programming paradigm, Æminium to develop by-default concurrent applications based on access permissions. Æminium supports `fork/join` and `dataflow` parallelism. It extends Plaid's compiler and runtime to support permissions as the first class language constructs. Like Plaid, it supports three kinds of permissions i.e., `unique`, `immutable` and `share` but unlike Plaid that uses access permissions to track the state of a typestate-based program, Æminium uses access permissions to avoid side effects in a program to parallelize its execution.

In Æminium, programmers explicitly specify permission-based annotations in the source program to express the read and write behavior of every object being accessed inside a method. The `unique` and `immutable` permissions are used to specify `exclusive` and the `read-only` access to the referenced object and the `share` permissions are used to specify concurrent access to the shared objects.

Æminium runtime leverages permission flow through the system (for methods' parameters or receiver objects) by extending the Plaid type checker and generates a permission dependency graph to computes data dependencies at the task level. Figure 2.6 shows the permission flow graph of the `transfer()` method given in Listing 2.2 in Æminium. This information is then used to parallelizes execution of the program to the extent permitted by the computed dependencies.

For the `transfer` method, the `unique` permission associated with `receiver` object flows into `deposit()` method while the `unique` permission assigned to `sender` object is transferred to the `withdraw()` method. However, the `transfer` method has only one `immutable` per-

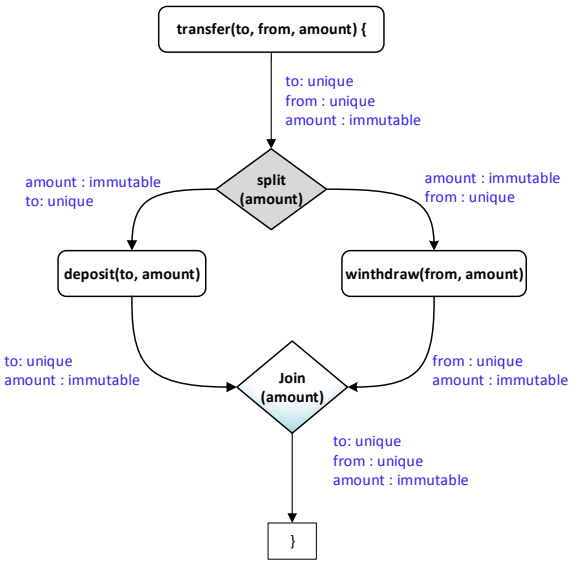

Figure 2.6: The permission flow of the transfer method in Æminium (Stork et al., 2014)

mission associated with the `amount` object which is automatically split into two `immutable` permissions by Æminium and fed into each of the method calls separately. In this way, both methods can be executed in parallel because both read and write different bank accounts i.e., `sender` and `receiver` and do not produce side effects on each other.

Æminium was able to achieve performance improvement over the sequential versions of the program based on access permissions but at the cost of high annotation overhead. Further, the performance gains in Æminium are limited due to the inherent performance bounds imposed by the Plaid language itself and its sophisticated type system.

However, in all the approaches discussed above, manually adding permission-based specifications in a program pose significant annotation overhead for programmers and that is itself a tedious and error-prone task to undertake. Hence, automatic inference of permission-based dependencies from the source code (the aim of the research presented in this thesis) is highly desirable to increase the wider adoption of existing verification and parallelization approaches and to employ them for the general-purpose program development and verification.

Chapter 3

Approach

This chapter explains the access permission inference approach in detail and elaborates the permission inference mechanism using a motivating example. It further provides an overview of the permission checking mechanism through the permission-based model checking tool Pulse. It further explains the theoretical background of the extensions made in the Pulse tool to perform a comprehensive concurrency analysis of sequential programs, based on the inferred permissions. The inference of access permission contracts and its proof-of-concept using the existing permission-based verification tool can be used to achieve implicit concurrency for the sequential programs, written in the imperative or object-oriented programming languages such as Java, without the fear of data races.

3.1 Overview

The permission inference approach automatically generates five types of access permissions for an un-annotated single-threaded Java program. Further, we integrate and extend the Pulse tool as a part of the permission checking mechanism. The inference approach reveals the implicit dependencies present between the code (methods) and the shared objects and maps them in the form of access permissions using graph abstractions. The permission checking mechanism in Pulse then verifies the correctness of the inferred specifications and computes the potential for concurrency in the source program.

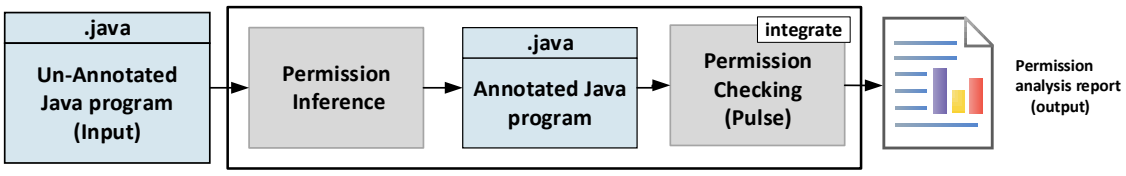

Figure 3.1: A high-level workflow of the permission inference and checking mechanism.

The approach generates five types of symbolic permissions for the object's (class) field thereby, generating permissions at a more granular level that can be used to exploit the maximum concurrency present in the code.

The permission inference and checking mechanism are explained in detail in Section 3.2 and Section 3.3, respectively.

3.2 Permission Inference

To generate access permissions for the shared objects at the method level, following the access permission semantics (Section 2.1), we need to identify the way (`read` or `write`) a referenced object[1] is accessed by the current method and, at the same time by its context ("the rest of the world except the current method"). Moreover, we need to identify and track aliases of the referenced objects (if any), to identify the correct dependencies and to maintain the integrity of the data during analysis. For this purpose, the inference approach performs modular static analysis of an un-annotated Java program based on its Abstract Syntax Tree (AST). The approach uses a set of pre-defined syntactic rules to distinguish between different type of expressions in the generated AST and to support the read-write and alias-flow analysis of the objects accessed in a method. The extracted (dependency) information is then used to generate access permissions on the referenced objects, following the pre-defined access permission inference rules. The permission inference approach takes the following steps or phases as shown in Figure 4.2:

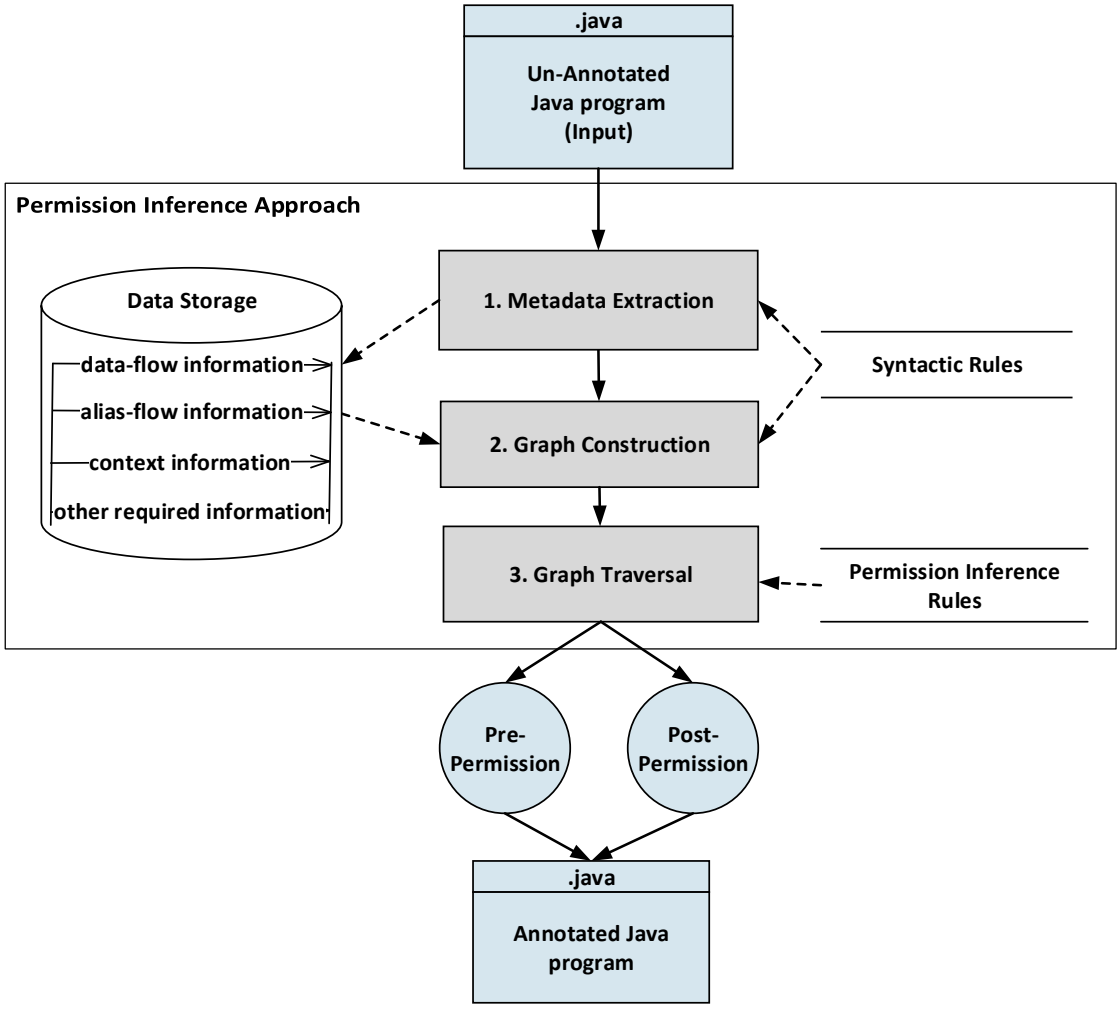

Figure 3.2: The permission inference approach.

[1]The term reference variable and reference object has been interchangeably used throughout the thesis, to represent an object accessed in a method.

Metadata Extraction. It parses AST of the Java program to extract and maintain the metadata (dependency) information as data-flow, alias-flow and context information, for all the objects (fields) accessed in a method from its global environment (Section 3.2.1).

Graph Construction. For each method, based on the extracted information, it constructs a permission-based graph model, using graph notations and by following the pre-defined syntactic rules (Section 3.2.2) that specifies the way to model object's accesses in a graph structure.

Graph Traversal. It traverses the constructed graph for each method and generates symbolic permissions for the object's (class) fields accessed in a method using access permission inference rules (Section 3.2.3).

The approach automatically generates five types of access permissions e.g., `unique, full, share, pure and immutable`), as `pre-` and `post-permissions` on individual field of an object (class) at the method level. It further generates an annotated version of the input program with permission contracts following the Plural specifications (2.2.1) where permission are defined on the (whole) referenced object. The pre-permissions are the permissions that caller (client) of a method must provide on the referenced object(s) before invoking a method or alternatively, the permissions that method requires on the referenced objects (fields, parameters etc.) before being executed. The `post-permissions` are generated on the referenced object(s) when the method completes its execution.

It is worth mentioning here that, following the Design by Contract principle, a method is responsible to return either the consumed (same) permission back to the caller of method, e.g., `unique` for `unique` or generate some restrictive permission such as `full` for `immutable`, as post-permission, to avoid the data integrity problems when permissions are actually used for verification or parallelization purpose. However, in certain cases, the pre- (post) permission on a referenced object could be some special case e.g., the `none` permission.

For example, in a Java program, it can happen in three situations a) if a method (constructor) instantiates the (global) referenced object in its local environment, b) if a method itself is the `main()` method from where execution of the program starts, c) if a method (constructor) creates a null reference or un-instantiates the (global) referenced object, or d) if a method reads an object that is not being accessed by other methods in any way. For the first two cases, the approach generates `none` as pre-permission, showing the absence of permission, which means that the method does not require any permission on the referenced object to start its execution. In the last case, the approach generates `none` as post-permission on the reference object.

Another special case is when a method reads a field an object (class) that is not shared across other methods in any way, in this case, the pre- and post-permission for the referenced object would be `none`. Moreover, no permission contract would be generated for a method if it does not access any object's (class) field from its global environment or if it manipulates only the local references declared in it.

In the following sections, we will elaborate on each of the three phases of the above-mentioned permission inference approach using the user-defined `ArrayCollection` program shown in Listing 1.1 and some methods from the benchmark programs that are used for the evaluation purpose.

3.2.1 Metadata Extraction

The approach performs data-flow, alias-flow and context analysis[2] of the source code to extract and maintain the read, write, aliasing information for all the object's (class) fields accessed in the current method and their access by the other methods. The extracted information is then used to compute the permission-based side effects at the method level. The analysis is based on the type of expressions[3] such as `<FieldAccess>` and `<MethodInvocation>` expressions encountered in an expression statement [4]. The analysis further depends on the type of reference variable referring to an instance (class) field, a method's parameter or a local reference that is an alias of some global reference. The analysis ignores the method's local variables and parameters that do not refer to any global object, this is because manipulating local variables does not affect the access rights of the current and the other methods.

Identifying Object's Accesses in the Current method

The data-flow and alias-flow analysis work in a way that, for each method in the input program, the technique parses the method's signature and its body to identify and track the object's accesses as read, write and aliasing information in following ways:

Method Signature: The analysis parses a method's signature with its name, return type, visibility modifier,and formal parameters. The formal parameters are mapped with their corresponding argument (aliases) objects by fetching the method invocations of the corresponding method. This information is then used to extract (maintain) the read, write and aliasing information of the actual objects against parameters and to avoid the data inconsistency problems while the same object is being accessed by other methods. For example, in Listing 1.1 for method `manipulateObjects()` in Line 56, the technique maps the formal parameters `p1` and `p2` with their actual objects i.e. `w` and `z` respectively and then track them in the method body to identify their metadata information.

Method Body: In parsing a method, the technique parses each expression statement in the AST of the method body. Each expression in an expression statement is iteratively parsed to distinguish (fetch) the `<read-only>` and `<read-write>` expressions and this information is recorded accordingly. Like parameters, the technique maps all the local references declared in a method with their global references (aliases) if any, to extract and maintain the data-flow and alias-flow information of the actual referenced objects during parsing.

The handling of the read and write expressions in an expression (statement) is as follows:

- The `<read-only>` expressions are characterized by expression nodes such as `<FieldAccess>`, `<QualifiedName>`, `<SimpleName>`, `<MethodInvocation>` etc., in the AST and parsed to extract all the object's (class) fields accessed in the expression. This information is maintained in the system, as a part of data-flow (read access) analysis, for the referenced objects

[2]It is worth mentioning here that, in this thesis, the meanings of context analysis is different from the term context-sensitive analysis. In our permission inference technique, context refers to other methods (except the current method) accessing the same object as the current method.

[3]The expressions are characterized by a node of type <Expression> in the AST.

[4]The expression statements are characterised by a node of type <ExpressionStatement> in the AST.

in the current method. For example, in Listing 1.1 in Line 61, the technique parses `p2.data` expression as a `read-only` expression. The information is maintained as read access for the actual object referenced by variable `z` in the current method `manipulateObjects()`.

- The `<read-write>` expressions are characterized by `<Assigment>` expression in the AST. The proposed technique performs flow-insensitive analysis of the source code, it ignores the order of execution of statements. However, the analysis preserves the semantics of assignment statements and precisely extracts the data-flow and alias-flow information of the referenced objects in an assignment statement, by determining the type of a reference on the left-hand side of an assignment statement based on its right-hand side expression. During parsing, the assignment expressions are further categorized as `<value-flow>`, `<address-flow>`, `<object-creation>`, `<null-address-flow>` or `<self-address-flow>` expressions, based on the type of the right-hand side expression and handled accordingly. For example, if the right-hand side expression yields a `<Primtive>` type or if it is a `<NumberLiteral>` expression, the approach treats the assignment expression as a `<value-flow>` expression. The `<object-creation>` expressions are characterized by the `<ClassInstanceCreation>`, `<ArrayCreation>`, `<ArrayInitializer>` expressions present on the right-hand side of an assignment expression. Similarly, If the right-hand side yields a `<ReferenceType>` or a `<NullLiteral>`, the expression is categorized as a `<address-flow>` or a `<null-address-flow>` expression respectively. The analysis recursively parses the right and the left side of an assignment statement to identify the expression type and extracts (maintains) the read, write and aliasing information of all the object's (class) fields accessed in each expression.
  Let us take Listing 1.1 again as an example to elaborate on the metadata extraction mechanism using different expression types.
  - The assignment expression `x = p1;` in Line 57 in method `manipulateObjects()` is categorized as an `<address-flow>` statement. This is because the right-hand side (`p1`) is a reference type and an alias of object `w` which means `x` is an alias of `w`. This information is maintained in the system as part of alias-flow analysis for object `w`, as any change in the state of object `x`, made directly or indirectly through any reference can affect `w` and its aliases. Further, the access for the referenced variable `w` is stored as read access by the current method `foo`.
  - Similarly, the expression `Client t = x;` in Line 58 in the same method is categorized as <address-flow> statement for the local reference `t` that now refers to a global reference `x`. The analysis tracks this information as a part alias-flow analysis between reference `t` and `x`, as any change in reference `t` can affect the state of the referenced object against `x` and vice versa. The access for the right-hand side operand i.e., `x` is handled as a `<read-only>` expression.
  - The expression `y = t;` in method `manipulateObjects()` in Line 59 is categorized as an `<address-flow>` expression. The analysis maps the local reference `t` with its global reference, as an alias of `x`. This information is maintained as a part of alias-flow analysis

for the actual object referenced by the variable `x` as any change in the state of `x` made directly or indirectly through `t` would change the state of its other aliases.

- The expression `x.data = 10;` in Line 60, is treated as a `<value-flow>` expression as the right hand side of the expression is a `<NumberLiteral>` constant. The information is updated as write access for the referenced object `x` in the current method. Further, the analysis ensures that this change (write operation) should be propagated to all the aliases of `x` i.e., reference `y` and `w` in this case, to ensure the integrity of data.
- The expression `array1 = new Integer [10];` in the `ArrayCollection()` method in Line 4, is an `<object-creation>` expression, following the right-hand side of the assignment expression i.e., `<ArrayCreation>`. The analysis maps the `array1` access as write access in the current method being the left-hand side of the assignment statement.
- Similarly, the approach treats the expression `this.array1 = null;` in Line 43 as as an `<null-address-flow>` statement and maps it as a write access for the referenced object. Further, the analysis ensures that this change should be propagated to all the aliases of reference `array1` to maintain the integrity of the data during parsing, and to avoid side effects when the method is actually parallelized based on the generated permissions.

- Further, the **`method calls`** in a method body are handled using `<MethodInvocation>` and `<SuperMethodInvocation>` etc., expressions in the AST. As a part of the modular analysis, the permission inference technique is recursively applied to every callee methods (a non-recursive method call) in the caller method. For this purpose, the current state of the caller method is saved and restored when all of its sub-methods have been parsed. The analysis of the caller method will not complete until the metadata of all of its sub-methods have been extracted and permission on their referenced objects have been generated.
  For example, in Listing 1.1 in Line 38, the approach first parses method call expressions `printColl()`, `isSorted()` and `findMax()` in the given order, to extract the read, write and alias information on the object referenced by the parameter `coll` in the called methods. The extracted information constitutes the metadata information for the caller method. The approach then uses this information to generate the necessary permissions for the caller methods.

- It is worth mentioning that the analysis does not parse a method invocation expressions with a **`recursive method call`** (eg. when a method calls itself in its body). This is because a recursive method call does not change the way the caller method (itself) accesses its referenced objects even through the recursive call expression, reducing the analysis time and the infinite loop as well. For this purpose, the analysis performs on-the-fly call graph analysis (Grove and Chambers, 2001; Lhoták and Hendren, 2003) to identify a direct or indirect recursive calls in the caller method.
  For example, Listing 3.1 shows a sample method `seqFib()` from the Æminium benchmark program `fibonacci`. The method computes fibonacci of a given number `n`. For expression, `if(n <= 2)` in Line 4, the approach extracts the metadata information of the referenced object `n` passed as a parameter. The approach does not parse the caller method again

when a recursive call `seqFib(n - 1)` and `seqFib(n - 2)` to itself encounters in Line 7 with parameter `n`, as parsing the caller method again will not change the way (read or write) object `n` is accessed in it.

Listing 3.1: A sample recursive method in Java.

```
1 public class SeqFibonacci{
2 Integer n = 5;
3  public static Integer seqFib(Integer n){
4  if(n <= 2)
5   return 1;
6  else
7   return (seqFib(n - 1) + seqFib(n - 2));
8  }
9 }
```

- Similarly, in case of **`infinite and chained recursion`**, (eg. when a method, say `foo1()` calls another method `foo2()` that in-turn calls `foo1()` in its body), the analysis does not parse method `foo1()` again, as a result of the second-level (indirect) recursive call, and analysis terminates successfully. This is because the approach maintains the current state and the metadata (at least signatures) of each method, say `foo1()`, before parsing a (sub-) method called inside the caller method which helps to identify and skip the indirect recursive call to the caller method itself. In this way, the analysis continues parsing from the next expression in the caller method say `foo2()`, if any, without creating an infinite loop during parsing. For example, Listing 3.2 shows a sample method `phi()` from a Java Grande benchmark program (`search`). The method `phi()` declared in Line 3 invokes method `Phi()` in Line 4 that in-turn calls the method `phi()` in Line 7. This is an example of indirect recursive method call. The approach does not fetch the `<MethodDeclaration>` of the method `phi()` to parse it again as it is currently being parsed (the parsing is already on its way). The approach only extracts the dataflow information of the objects referenced by the class fields `z, mu, sigma` in Line 7, being arguments in the method call `phi()`, as a read access by the current method `Phi()` and control returns back to the caller method `phi()`. T here are no more statements in method `phi()`, the analysis terminates without creating any loop during parsing.

Listing 3.2: A sample code snippet for indirect recursion and method overloading in Java.

```
1 public class Gaussian {
2  public static double z = 0.2, mu = 0.3, sigma = 0.4;
3  public static double phi(double z){
4   return Phi((z - mu) / sigma) / sigma;
5  }
6  public static double Phi(double z){
7   return phi(z - mu) / sigma) / sigma);
8  }
9  public static double phi(double x, double mu, double sigma){
10  return Math.exp(-x * mu / sigma) / Math.sqrt(2 * Math.PI);
11 }
12 }
13 // test client
14 public static void main(String[] args){
15  Gaussian.phi(z);
16  Gaussian.phi(z, mu, sigma);
```

- However, the **`static method dispatching`** to multiple targets is straightforward and is determined based on the method signature and the static type of the referenced object at compile time.

For example, in Listing 3.2, the approach maps the method call expression in Line 15 with the method signature `phi(double z)` in Line 3 to extract the object's accesses in it. Similarly, the method call in Line 16 is mapped with its method declaration `phi(double x, double mu, double sigma)` in Line 9, to parse its body.

- The **`super method calls`**, in case of `inheritance`, are handled the same way (parsing a method's signature and its body) as other non-recursive method calls. These expressions are captured using the `<SuperMethodInvocation>` or `<SuperConstructorInvocation>` expression type in the method body and parsed accordingly.
  For example, Listing 3.3 shows a parent-child class hierarchy from the `montecarlo` program in the Java Grande benchmark. The child class `ReturnPath` calls the a parent method, using expression `super.dbgDumpFields()`, in Line 4. The analysis parses the body of the method `dbgDumpFields()` by fetching its `<methodDeclaration>` node. It extracts the dataflow information for data members `DEBUG` and `prompt` in Line 10 which is then mapped as `read` access for the current (child) method in Line 4. This information is then used to generate access permission rights for the callee and caller method.

Listing 3.3: A sample code snippet to eleborate inheritance and super constructor (method) calls in Java

```
1 public class ReturnPath extends PathId {
2  public boolean DEBUG = true;
3  protected String prompt = "ReturnPath>";
4  public void dbgDumpFields(){super.dbgDumpFields();} //super method call
5 }
6 public class PathId{
7  public boolean DEBUG = false;
8  protected String prompt="PathId>";
9  public void dbgDumpFields() {
10   System.out.println("debug="+this.DEBUG+"prompt="+this.prompt);
11  }
12 }
```

- For **`library method calls`**, the approach generates the safe access, (write instead of read) as a conservative approach, for the referenced object even if it is only read in the method body to maintain the integrity of data if the state of the object is updated by the library method call. This is because of the unavailability of the method's definitions of the library methods in the source code.
  For example, in Listing 3.4, the class `ExceptionalLabel` declares a method `getHascode()` in Line 3 and calls the `hashcode()` method of a reference type `ITypeBinding` in Line 5, through the object reference `exceptionType`. Although, it only reads the object reference `exceptionType` in Line 5. However, our approach maps the method's operation, in this case, as write operation to generate safe permissions afterward and to maintain the integrity of the data.

Listing 3.4: A sample code snippet for a class library method call in Java

```
1 public class ExceptionalLabel implements ILabel {
2 private ITypeBinding exceptionType;
3 public int getHascode() {
4  if(exceptionType == null){ return 0; }
5  else{ exceptionType.hashCode(); }
6 }
```

- The **`conditional and dynamic structures`** in the source program such as **`switch-cases, if-else, and loops statements`** do not affect the permission inference mechanism as the approach parses all expressions encountered in an expression statement, based on the type of expression and the type of object accessed in it irrespective of its access location. The analysis ensures to update the metadata of a reference variable every time it is accessed in different expressions (including the conditional or dynamic structures), thereby extracting the safe access and alternatively, generating the restrictive permissions.
  For example, Listing 3.5 shows a sample method `setAliasPerObject` in the class `PulseSettings`. The method reads the reference variable `aliasPerObject` in Line 4 (`if` part) but it writes on the referenced object in Line 6 (`else` part) of the `if-else` statement. The analysis extracts the write access on the referenced object as safe access.

Listing 3.5: A sample code snippet for the conditional statements in Java

```
1 public class PulseSettings{
2 int aliasPerObject = 0;
3 public int setAliasPerObject(int x){
4  if (x <= 1){
5   return aliasPerObject;}
6  else{
7    aliasPerObject = x;
8   }
9   return aliasPerObject;
10 }
11 }
```

- Moreover, handling of the **`array data structures`** is the same for single- and multi-dimensional arrays. This is because,at the moment, the approach does not generate permissions on the individual elements (dimensions) in an array data-structure. It parses the whole array object like an ordinary instance (class) object. The metadata analysis for one-dimension arrays, using the `ArrayCollection` class shown in Listing 1.1, is already explained in this chapter.
  Listing 3.6 shows a bit complex example of the array data structures with aliasing between arrays of different dimensions. The example method `dgefa()`[5] is taken from the case study `lufact` in the Java Grande benchmark. It manipulates a one- and two-dimensional array using object's fields `ipvt` and `a`. respectively, passed as parameters. The approach extracts the data-flow and alias-flow information for both objects to generate permission contract for the current method.
  The approach first parses the method's signature with its parameters `a` and `ipvt` and stores this information as a part data-flow analysis (read access) for both the objects in the current method. While parsing method body, the approach ignores the method's local declarations expressions in Line 3-6 as well as the local variables accessed in conditional and iterative statement in Line 7 and 8, as none of those affects the way the method accesses object `a` and `ipvt`. The expression `(col_k = a[k];)` in Line 9 is treated as an `<address-flow>` statement. This is because the method's local reference variable `col_k` starts pointing to the object's field `a` using its one dimension. This information is maintained as a part of

[5] We removed some of the code from the example method, while keeping all the operations intact, to focus on the array objects in the method.

alias-flow analysis for object `a`, as any change in the state of `a` through `col_k` should be considered to identify the correct dependencies at the method level, and to maintain the integrity of the data during parsing and afterward.

Listing 3.6: A sample (code snippet) to elaborate handling of the iterative structure and two dimensional arrays in Java

```
1 public class Linpack {
2  final int dgefa( double a[][], int lda, int n, int ipvt[]){
3  int k,kp1,l,nm1,t = -1;;
4  int info = -1;
5  nm1 = n - 1;
6  double[] col_k;
7  if(nm1 >=  0){
8   for(k = 0; k < nm1; k++) {
9    col_k = a[k];
10   kp1 = k+1;
11   if(col_k[l] != 0){
12    t = -1.0/col_k[k];
13    dscal(col_k,n,t,kp1,1);
14   }
15  }
16  else{
17  ipvt[n-1] = n-1;
18  info = k;}
19  }
20  return info;
21 }
22 final void dscal(double dx[], int n, double da,  int dx_off, int incx){
23 if(incx != 1) {
24  for (i = 0; i < n; i++)
25   dx[i +dx_off] *= da;
26 }
27 }
```

The analysis ignores the expression in Line 10 as manipulates the method's local variables only. The conditional expression and the mathematical statement following it in Line 11 and 12 are parsed as `<read-only>` expressions for reference object `a`. This is because the approach maps the local reference `(col_k)` with its actual reference `a`.
In Line 13, the method call `dscal()` carries a reference to object `a`, using the reference `col_k` as its argument and the control is transferred back to Line 22 where the reference `col_k` is mapped with parameter `dx`, to be used by the method `dscal()`. The analysis again ignores the conditional expression in Line 23 and the loop statement following it in Line 24, as both manipulate method's local variables.
Further, the expression `(dx[i + dx_off] *= da;)` in Line 25 is treated as a `<value-flow>` expression following the `<Assignment>` expression in it. The approach updates this information as write access for the object referenced by parameter `dx` i.e., `a`. There is no other statement (expression) in method `dscal`, the control returns back to the method `dgefa()` following the method invocation expression `dscal()` in Line 13. Similarly, the data-flow information for the array object `ipvt` is updated as write access in the current method following the `<value-flow>` statement in Line 17. The analysis ignores the read-write operations on local variables in Line 18 to 20. The context analysis of the array objects `a` and `ipvt` in this method depends on their read and write access by other methods in the program.

Identifying Object's Accesses in Other Methods

The **context analysis** of all the objects accessed in the current method is based on their access by other methods in the program. There can be three possible contexts namely `Context-N` (no access), `Context-R` (read-only access) or `Context-RW` (read and write access) for a shared object. The `Context-N` is the most restrictive context as in this context, the current method demands exclusive rights (both read and write) on the object; thus reducing the chances to achieve maximum parallelism across methods. The read context (`Context-R`) is less restrictive than `Context-N` as it allows some kind of read access to other references thus providing the possibility to achieve more parallelism. `Context-RW` is the most flexible context as other references would have both read and write access to the referenced object but the possibility of parallelism is limited due to the expected (undesirable) effects, such as data races.

The approach automatically identifies the context information (**read, write and none**) for all the shared objects following their data-flow and alias-flow information in the program. It extracts the safe `context` (access by other methods), for the objects accessed in the current method, by updating their accesses across other methods. For example, in Listing 1.1, for method call `printColl(coll)` in Line 38, the approach generates `Context-RW` for the reference `array1`, if the method is called through the method call `computeState(obj1.array1)` in Line 69, as a part of its context analysis. This is because the array object is being written by other methods such as `incrColl()` and `tidyupColl()` in the program. The exceptions to this rule are the objects accessed in expressions categorized as `<object-creation>` and `<null-address-flow>` expression where current method (un)instantiates an object in its body. In this case, the approach generates `Context-N` (no-access by other methods) for the reference variable accessed on the left-hand side of an assignment expression, by following the expression type of the right-hand side expression.

For example, in Listing 1.1, for expression `(this.array1 = null ;)` in Line 43, the approach generates `Context-N` as its context information for `array1`. This is because the current method un-instantiates the referenced object and it should have exclusive access to create a null-reference (a reference variable that does not refer to any object) and update this information to all of its alias(es).

Similarly, for the `<object-creation>` expression such as `array1 = new Integer [10];` in Line 4 in Listing 1.1, the approach generates `Context-N` for array object `array1`, as the current (constructor) method `ArrayCollection()` instantiates a new object and, at this moment, no other method can access it. This information is then used to construct a permission-based graph model of the current method and generate pre- and post-permissions for the referenced objects accessed in it. The type of access permissions generated in each context depends on the way (read or write) the current method accesses the shared object.

It is worth mentioning here that in a Java program execution starts from the `main` method, therefore execution of the `main()` method is independent of any context which means it does not require any (pre-) permission to access the objects from its global environment. The approach ensures to generate `Context-N` (none) for all the referenced variables accessed in the `main` method. Following, the permission semantics (2.1), application of `Context-N`

in-turn generates unique permissions the referenced objects.

3.2.2 Graph Construction

In this section, we elaborate the graph construction phase and how we map programming constructs in the form of permission-based graph model. The graph construction phase models the read, write, aliasing and context information extracted in the metadata extraction phase in the form of a graph structure using graph abstractions. The generated model is then used to automatically reveal access permission-based dependencies in the source program. Figure 3.3i shows the permission-based graph model of method `manipulateObjects()` generated in this phase.

I. **Graph Notations**

The approach defines some special nodes and edges to model the programming constructs in a graph model which are described below:

Graph Nodes: The permission graph of each method is modelled using three types of graph nodes:

- A **`variable`** node, depicted as a labelled circle, models a reference variable to represent the **`object`** accessed by the current method and its context. For example, `x` represents an object referenced in method `manipulateObjects()` in Figure 3.3i.
- A method node, **`this_m`** is an abstraction, a labelled rectangle, that represents the **current method** accessing the referenced object. For example, method `manipulateObjects()` in Listing 4.1 is labelled as `this_m` in Figure 3.3i.
- A method node, **`context`** is an abstraction, depicted as a labelled rectangle, that represents **other methods** accessing the same object or collectively the current method's global environment.

Graph Edges: There are two types of **edges** that model the way (read, write, and alias), a referenced object (`o`) is accessed in the current method **(`this_m`)** and by its **context** (other methods).

- **read/write edges** are depicted as directed and solid edges labelled as 'r' or 'w'. For example, in Figure 3.3i, there exists a read and a write edge between method `this_m` and variable node `x`. Similarly, objects `(x, y, w, z)` are read by the client method (`main()`) in Listing 4.1, so a read edge with label 'r' is drawn from `context` to all the `variable` nodes.
- **alias edge** models an alias of a reference if any. The alias edge is depicted as a directed and dotted edge labelled with the letter 'a' between two (reference) variable nodes. For example, `x` is an alias of `w` in Figure 3.3i.

The nodes `this_m` and `context` have been introduced to make graph construction and traversal process simpler.

II. **Rules for Modelling the Object's Accesses in a Permission-based Graph**

We mathematically specify syntactic rules to support the metadata analysis of the source code and model the object's accesses in a permission-based graph model to generate permissions on the referenced objects. The rules are based on the type of expressions encounter in each expression statement. We have divided the rules into two main types depending on their usage: a) `Context` rules, b) `Statement` rules, during the analysis. The statement rules, for simplicity, are categorized as `method` call and non-method call rules depending on the type of expression encountered. The statement rules are further categorized based on the type of reference (variable) encountered in each expression statement.

For example, in the syntactic rules, the notation `<grv>` specifies a reference variable (`<grv>`) that refers to a (global) object's (class) field declared outside the method body, or a parameter that is an alias of the `<grv>`, the notation `<lrv>` represents a method's local reference that is an alias of `<grv>` and `<lv>` represents a method's local variable other than `<lrv>`.

Table 3.1 shows the conventions used to mathematically specify the graph modelling rules. The notations `<grv>` and `<this_m >` are being used as an example to show how specifications work to model a referenced object, `<grv>`, following its access (read, write and alias) information in the current method `this_m`. These conventions are equally applicable to model other methods (`<context >`) accessing the shared object and other types of reference variables such as `<lrv>` and `<lv>`, encountered in an expression statement.

The rules follow the style of sequent calculus in Linear logic, with logic connectives and implication ($\multimap$) operator, that considers rules as formulas (resources) and enforces their constructive interpretation to extract and map the object's accesses in a precise way.

Although the rules are self-explanatory, we explain some of them, to provide intuition on mathematically specifying the rules and their role in supporting the permission inference mechanism. A complete list of syntactic rules is given in Appendix 1.

A. Context Rules model the read, write behavior of other methods on the objects accessed in the current method. The context rules specify different ways to add read and write edges between the `context` and `variable` nodes. The rules are designed to follow the style of sequent calculus, as shown in Equation 3.4, where the rule name itself shows the type of context (`Context-N`, `Context-R` or `Context-RW`) applicable on the referenced object `<grv>`.

$$\frac{\texttt{<reference-variable>}}{\texttt{<Rule-Description>}}\texttt{(<Rule-Name>, <grv>)} \tag{3.1}$$

For example, the **`(Context-RW, <grv>)`** rule specifies that we need to add a read and a write edge from `context` to variable (`<grv>`) node to show that the object (`<grv>`) accessed in the current method is also updated by other methods. Similarly, the **`(Context-N, <grv>)`** rule specifies the absence of context for a referenced object `<grv>` which means the current method is the sole reference to `<grv>`. Therefore, we need to remove both read and write edges, if they exist, from `context` to `<grv>` node.

Table 3.1: Modelling Conventions.

Conventions	**Description**
addReadEdge(`this_m`, `<grv>`)	A function call to draw a read edge from the current method (`this_m`) node to a variable node (`<grv>`).
addWriteEdge(`this_m`, `<grv>`)	A function call to draw a write edge from the current method (`this_m`) node to a variable node (`<grv>`).
removeReadEdge(`this_m`, `<grv>`)	A function call to remove a read edge (if already exists) from the current method (`this_m`) node to a variable node (`<grv>`).
removeWriteEdge(`this_m`, `<grv>`)	A function call to remove a write edge (if already exists) from the current method (`this_m`) node to a variable node (`<grv>`).
addAliasEdge(`<grv>`, `<grv1>`)	A function call to add an alias edge from a `<grv>` node to another variable `<grv1>` node.
removeAliasEdge(`<grv>`, `<grv1>`)	A function call to remove existing alias edge from a `<grv>` node to another `<grv1>` node.
aliasEdge(`<grv>`, `<grv1>`)	A function call that checks if `<grv>` is an alias of `<grv1>`.
aliasOf(`<grv>`)	A function that returns all the aliases of an object referenced by `<grv>`.
apply(<Rule-Name>, `<grv>`)	This function specifies the application of a particular rule `Rule-Name` on reference variable (`<grv>`).
`<Type>`	It represents the data type (both reference and primitive types) of the referenced object unless specified as a `Primitive` or a `Reference` type using notations `<REF_TYPE>` or `<PRIM_TYPE>`.

$$\frac{\texttt{<grv>}}{\texttt{addReadEdge(context,<grv>), addWriteEdge(context,<grv>)}}\ \textbf{(Context-RW, <grv>)}$$

$$\frac{\texttt{<grv>}}{\texttt{removeReadEdge(context,<grv>), removeWriteEdge(context, <grv>)}}\ \textbf{(Context-N, <grv>)}$$

B. Statement Rules describe different ways (read and write) to add edges between a `variable` node and the current method. The statement rules for non-method call expres-

sions are designed to follow the style of sequent calculus, as shown below.

$$\frac{\texttt{<Exp-Statement>}}{\texttt{<Rule-Description>}}\,(\texttt{<Rule-Name>, <grv>}) \tag{3.2}$$

The antecedent `<Expression-Statement>` part in Equation 3.4 captures different types of the expression statements (`read-only, value-flow and address-flow`) and the consequent `<Rule-Description>` part specifies how to model the object's accesses in the current method and its context (if any).
The rule's name (`<Rule-Name>, <grv>`) itself follows the type of expression encountered during parsing and the type of reference variable (GR for `<grv>`, LR for `<lrv>` and L for `<lv>`) accessed in each expression.
For example, the **(GR-Read-Only, <grv>)** rule models the read access of the current method on the referenced object in case of a `<read-only>` expression. The rule states that we should add a read edge from the current method `this_m` node to the `<grv>` node.

$$\frac{\texttt{<grv> |super.<grv> |this.<grv> |<ClassName>.<grv> |<obj>.<grv>}}{\texttt{addReadEdge(this\_m, <grv>)}}\,(\textsf{GR-Read-Only,<grv>})$$

Similarly, the **(GR-Val-Flow, <grv>)** rule models the write access of the current method (`this_m`) on the object referenced by `<grv>` in case of a `<value-flow>` statement. It states that we should add a write edge from the current method `this_m` node to variable (`<grv>`) node. It further ensures that this change should be propagated to all the aliases of `<grv>` to maintain the integrity of data during parsing. Therefore, in the graph, we need to add a write edge from the `this_m` node to all its alias(es) nodes, if any. All the objects accessed on the right-hand side of an assignment statement are modeled as `<read-only>` expressions following the appropriate syntactic rule.

$$\frac{\texttt{[PRIM\_TYPE] <grv> = <grv1> |<LITERAL>}}{\begin{array}{l}\texttt{(addWriteEdge(this\_m, <grv>)}(\forall a \in \texttt{aliasOf(<grv>)} \multimap \texttt{(addWriteEdge(this\_m, a))),}\\ \texttt{apply(GR-Read-Only, <grv1>)}\end{array}}\,(\textsf{GR-Val-Flow,<grv>})$$

The **(GR-Add-Flow, <grv>)** rule models expression of the form `<grv> = <grv1>`. The rule states that we should add an alias edge from `<grv>` to `<grv1>` node that shows a `pointer-pointee` relationship between the underlying objects at the code level, and should remove the existing alias edge from `<grv>` to `<grv2>` node (if any). This information is maintained as a part of alias-flow analysis during parsing.

$$\frac{\texttt{[<REF\_TYPE>] <grv> = <grv1>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<grv>, <grv2>)} \multimap \texttt{removeAliasEdge(<grv>, <grv2>)}\\ \texttt{addAliasEdge(<grv>, <grv1>),apply(GR-Read-Only, <grv1>)}\end{array}}\,(\textsf{GR-Add-Flow, <grv>})$$

The **(LR-Addr-Flow, <grv>)** rule models `<address-flow>` expressions of the form `<lrv> = <grv>`. The rule states that we should add an alias edge between a local reference (`<lrv>`) node and the (global) reference (`<grv>`) node and should remove its existing alias

edge with other nodes if any. The approach keeps track of the changes in the state of <lrv>, as a part of the alias-flow analysis, that could affect the object referenced by <grv> and its aliases. Further, we follow the (GR-Read-Only, <grv>) rule for the reference <grv> on the right-hand side of the expression, to model its read access by the current method.

$$\frac{\texttt{[<REF\_TYPE>] <lrv> = <grv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv1>)}\multimap\texttt{removeAliasEdge(<lrv>, <grv1>)),}\ \texttt{addAliasEdge(<lrv>, <grv>),apply(GR-Read-Only, <grv>)}}\ \textbf{(LR-Addr-Flow, <grv>)}$$

C. Method Call Rules capture the method invocation expressions in an expression statement. Following modularity of the analysis, the approach ensures to extract and model the access permission information for the called (sub) method, before completing the inference mechanism for the caller method.

The method call rules specify the way to add read and write edges in the caller method graph as a result of a method call. The type of edges added in the caller (method) graph depends on the post-access permissions generated by the called method on its referenced object(s). This is because, when actually parallelizing code based on permissions, the caller method needs to provide the pre-permissions for all the object's (class) fields accessed in the called method, to execute it as a part of its body.

These rules are specified following the syntax given in Equation 5.2 where the antecedent part represents a method call expression such as MCall([<args>]) with or without argument(s), and the consequent part describes the way to model the object's accesses in the caller graph. The <Rule-Name> itself shows the type of access permissions (<post-perm>) generated by the called method on its referenced objects (<grv>).

$$\frac{\texttt{<MCall-Expression>}}{\texttt{<Rule-Description>}}\texttt{MCall(<post-perm>, <grv>)} \tag{3.3}$$

The **MCall(<Pure>, <grv>)** rule generates pure permission, as a post- permission, on the object referenced by <grv>. It states that we should add a read edge from reference variable (<grv>) node to this_m node, to represent its read access by the current method, and we should apply (Context-RW, <grv>) rule to represent its read and write access by other methods. Similarly, for a method call **MCall([<args>])** that generates full permission on the referenced object <grv>, where the argument (args) can be an alias of <grv> or it can itself be the reference <grv>, as post-permissions. In this case, following the semantics of full permission, we need to add both a read and write edge from the called method (this_m) to the reference (<grv>) node, and should apply (Context-R, <grv>) rule on <grv> to model its read access by other methods.

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addReadEdge(this\_m, <grv>), apply(Context-RW, <grv>)}}\ \textbf{MCall(Pure, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addWriteEdge(this\_m, <grv>), apply(Context-R, <grv>)}}\ \textbf{MCall(Full, <grv>)}$$

The method producing other kinds of permissions on the objects referenced by `<grv>`, are represented similarly. Figure 3.3 shows the graphs generated for all the methods in Listing 1.1.

It is worth mentioning here that in the generated graph models, all the parameters are mapped with their actual (global) references, using the method invocation expressions in the program, to generate access permissions on the actual referenced objects afterward.

3.2.3 Graph Traversal

In the graph traversal phase, the approach traverses the graph model of each method to generate access permissions on the objects accessed in a method. The access **permission inference** rules generate five kinds of symbolic permissions and a special permission, i.e., `none` where the type of access permissions generated depends on the type of edges between the current method (`this_m`), and the reference `variable` nodes, and the presence (or absence) of alias edges between the `variable` nodes in the constructed graph.

The access permission inference rules follow the style of sequent calculus, as shown below, where the antecedent part (`<Rule-Description>`) describes the graph traversal steps and the consequent part (`<permission>, <grv>`) shows the type of symbolic permissions generated on the referenced object `<grv>`.

$$\frac{\texttt{<Rule-Description>}}{\texttt{(<permission>, <grv>)}}\ \texttt{(<Rule-Name>)} \tag{3.4}$$

- For example, the **Pure** rule guarantees that the shared object `<grv>` is only being read by the current method (`this_m`), but other methods (`context`) may have read (write) access on it, complying with the definition of the `pure` access permission given in Section 2.1. This rule states that:

  - There must not be a write edge from `this_m` to `<grv>` node; and
  - There must exist a read and a write edge from `context` to `<grv>` node.

- Similarly, the **Full** inference rule states that the current method (`this_m`), can only write on the referenced object `<grv>`, but other methods (`context`) can only read it. This rule generates `full` permission on `<grv>` if:

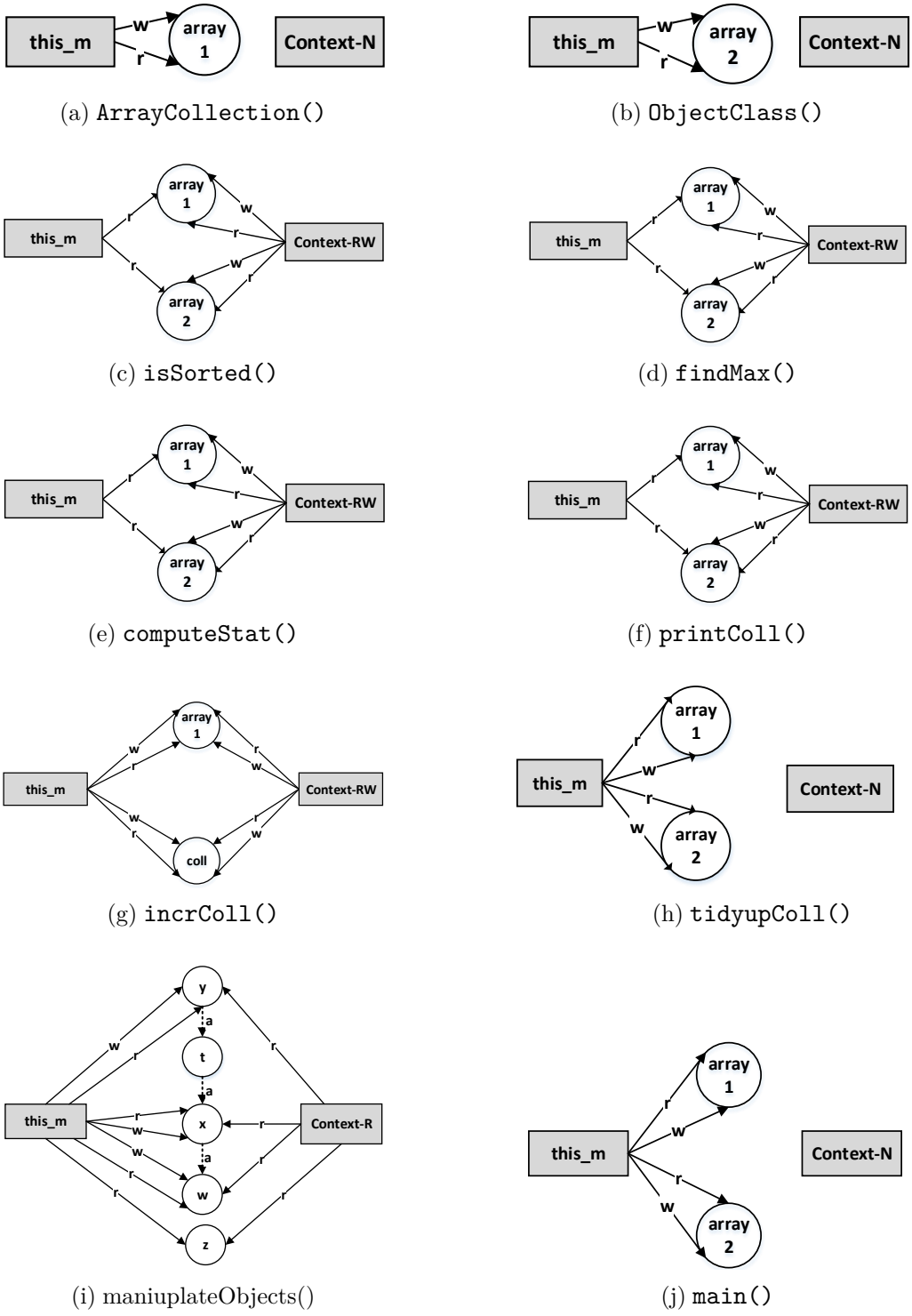

Figure 3.3: Permission-based graph models for the methods shown in Listing 1.1

.

- There must exist a read and a write edge from `this_m` to `<grv>` node.
- There must not be a write edge from `context` to `<grv>` node; and

$$\frac{\exists \texttt{readWriteEdge(this\_m,<grv>)} \wedge \exists \texttt{readEdge(context,<grv>)} \wedge \neg \exists \texttt{writeEdge(context,<grv>)}}{\texttt{full(<grv>)}} \textbf{(Full)}$$

$$\frac{\neg \exists \texttt{writeEdge(this\_m, <grv>)} \wedge \exists \texttt{readEdge(context, <grv>)} \wedge \exists \texttt{writeEdge(context, <grv>)}}{\texttt{pure(<grv>)}} \textbf{(Pure)}$$

For example, Listing 3.7, line 1 & 2 shows the pre- and post-permissions generated for the method `manipulateObjects()` by traversing its graph model in Figure 3.3i and by following the full and immutable permission inference rules on the referenced objects.

Listing 3.7: The permission contract for the method manipulateObject() using the Plural syntax

```
1 @Perm(requires="full(x) * full(y) * full(w) * immutable(z)",
2       ensures="full(x) * full(y) * full(w) * immutable(z)")
3 Client manipulateObjects(Client p1, Client p2){ ...}
```

The complete annotated version of the input program given in Listing 1.1 is shown in Listing 4.1 in Plural format with permission contracts generated at the field level. A complete list of access permission inference rules is given in Appendix A.3.

3.3 Permission Checking

The second part of the permission inference approach comprises a permission checking mechanism. For this purpose, the approach integrates Pulse, a model-checking approach implemented as Java Eclipse plug-in. Moreover, the permission inference approach extends the Pulse tool to capture all possible side effects based on five types of access permissions and to perform a comprehensive concurrency analysis of the input sequential programs, based on the inferred specifications.

The details of the permission checking mechanism in Pulse are already explained in Section 2.2.2. Pulse takes a Plural annotated program i.e., a Java program annotated with access permission contracts and typestate information, as input. It automatically verifies the correctness of the inferred specifications along with some of the program properties. Moreover, it analyzes the input specifications for concurrent execution that can help enable potential concurrency present in the sequential programs.

3.3.1 Correctness and Concurrency Analysis in Pulse

The permission checking approach in this research employs Pulse analysis to verify the correctness of input specifications and to perform their concurrency analysis in following ways.

Access permission Correctness. Pulse enforces that access permissions do not violate their intended semantics, by defining a discrete state semantics of the input specifications and by verifying the correctness of the input specifications, following the CTL formula given in Equation 2.9 and 2.10. Further, Pulse performs method (un)satisfiability analysis of the input specifications to identify missing specifications following the CTL formula defined in Equation 2.11.

Method Satisfiability Analysis. The method satisfiability (reachability) analysis of a method in Pulse is based on the `requires` clause (pre-permission) in a method contract. Pulse uses the $satisfiability_i(m)$ predicate in Equation 2.11 to check whether the `pre-condition` of a method `m` is met. A method is satisfied (reachable) if all its pre-conditions are met or if it gets enough (pre-) permissions to start its execution.
The presence of the unsatisfiable methods, due to the method's unsatisfiable pre-condition, indicates an error (misspelled or missing specifications) in the input specifications, or it can be due to program error such as the use of `null` references. The presence of an unsatisfiable method indicates that no possible client can fulfill the method's contract i.e., the `requires` clause and this method is not called under any circumstances; thus the method remains unreachable.
The Pulse correctness and method satisfiability analysis for the example program given in Listing 1.1, is shown in Section 4.3.1 that represents the efficacy of the proposed approach in generating the correct specifications without any specifications errors.

Concurrency Analysis. Pulse computes the possible number of method pairs that can be executed with each other in a program based on the access permission contracts. It does not consider the control-flow dependencies between method calls therefore, it computes an over-approximations (superset) of the number of methods that can be executed with at least one other method, at the class level, by following the (pre-) permission contracts between two methods.
The concurrency analysis of the input specifications in Pulse is based on the predicate $concurrent(m_1,m_2)$ as given in Equation 2.12. The concurrency analysis in Pulse identifies whether two methods can be executed in parallel including a method with itself, by identifying the immutable methods, i.e., methods that require read-only access on the shared object and the independent methods, i.e., methods that do not touch the same global object.

3.3.2 Extensions made to the Pulse Concurrency Analysis

The permission checking approach extends the Pulse analysis to perform a comprehensive concurrency analysis of sequential Java programs. In Pulse, two methods are considered parallel if both require `pure` (read access) as pre-permission on the referenced object. The non-parallel behavior of the two methods is determined based on `unique` and `full` permissions. In Pulse, the methods that require read-write (`unique` or `full`) access cannot be parallelized with each other.

However, the Pulse tool itself is limited in three ways a) Pulse tool does not support

overloaded methods (including constructors) even if the method is provided with the different return type, parameters, and permission contracts, b) it does not consider all the possible side effects comprising `full` and `pure` permissions as a part of its concurrency analysis, and c) it does not support `immutable` and `share` permissions, as a part of its correctness and concurrency analysis, and consequently, the model-checker is not able to consider the method's side effects for following pairs of (pre-)permission contracts between two methods e.g., `{(full, pure), (pure, full), (share, pure), (share, share)}`, and reports them to be concurrent. However, methods with these specifications, as a part of the method contract, if allowed to execute in parallel can cause data races of the form `<write-read>`, `<read-write>` or `<write-write>`. Table 3.2 shows the expected method-pair concurrency analysis that should be performed for a sequential program to void data races when the program is actually parallelized based on access permissions. The symbol $\parallel$ indicates the parallel execution of the two methods, whereas the symbol $\nparallel$ shows the fact that the specified methods should be executed in parallel with each other.

Table 3.2: Expected method-pair concurrency analysis.

		APs(m2)				
		unique	full	share	immutable	pure
APs(m1)	unique	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$
	full	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$
	share	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$	$\nparallel$
	immutable	$\nparallel$	$\nparallel$	$\nparallel$	$\parallel$	$\parallel$
	pure	$\nparallel$	$\nparallel$	$\nparallel$	$\parallel$	$\parallel$

In Pulse, the model-checker does not perform concurrency analysis of the program using pre-permission contracts of the form `(immutable, immutable)` and `(pure, immutable)` between two methods. Table 3.3 shows the method-pair concurrency analysis performed by the Pulse tool where the symbol ✓ indicates the options where Pulse identifies the method's side effects correctly, whereas the symbol ? shows the option where either the Pulse too does not support the permission annotations or performs incorrect analysis. For example, recall the Pulse concurrency analysis for the MTTS program (Section 2.2.2 in Table 2.3), where it reports method `setData()` and `getData()` to be concurrent but the parallel execution of these methods can create a race condition of the form `<read-write>` or vice versa.

The `immutable` permission being the safe permission, if applicable, can support maximum parallelism between methods without the fear of data races, and `share` permission being the most flexible access can create side effects thereby, data races. Therefore, these specifications should be considered, as a part of the concurrency analysis, for sequential programs to parallelize their execution without the fear of data races.

Our permission inference approach extends the Pulse concurrency analysis by considering all the possible side effects at the method level, based on five types of access permissions. The

Table 3.3: Method-pair concurrency analysis in Pulse.

		APs(m2)				
		unique	full	share	immutable	pure
APs(m1)	unique	✓	✓	?	?	✓
	full	✓	✓	?	?	?
	share	?	?	?	?	?
	immutable	?	?	?	?	?
	pure	✓	?	?	?	✓

objective was to compute the potential for concurrency in a sequential program based on the access permission contracts. The extended concurrency analysis can be used to parallelize the execution of Java programs, to the extent permitted by the inferred dependencies, without the fear of data races.

Recall the Pulse analysis in Section 2.2.2, Pulse first translates the input specifications into a (semantically) equivalent discrete state semantics. It then encodes the generated semantics in a state-machine model to be verified by the state-of-the-art evmdd-smc symbolic model-checker (Roux and Siminiceanu, 2010). The model-checker then verifies the core program properties such as the absence of deadlocks, missing specifications, the correctness of specifications and concurrency among methods, by performing the reachability graph analysis of the generated model, based on the pre-defined CTL formulae.

Our permission checking approach extends the Pulse concurrency analysis, to consider the method's side effects for five types of symbolic permissions, in the following ways:

1. **Discrete State Semantics.** The approach generates the discrete state semantics of the input specifications where.

- It generates a permission (fractional) sharing model (given in Table 3.4) that defines new token distribution for the current r_i^j and other reference r_i^l, in case of write tokens. The objective is to make the side effects (the write access) of the current method explicit, in case of `unique, full, share, immutable and pure` permissions, for other methods accessing the same object. The new model modifies the permission sharing model in Pulse given in Figure 2.3. The new model designed in a way that method requiring write access on the shared object should always run in isolation.
  In Table 3.4, the column $f(this)$ represents the number of read-write tokens that the current reference r_i^j requires, to fire a transition corresponding to the start of a method. The column $f(bank)$ defines the number of tokens that should be present in the bank i.e., enabling condition that should hold for a transition to fire, whereas $f(bank\prime)$ shows the number of read-write token in the back, corresponding to an (`update expression`), once the method has started its execution with reference r_i^j and the corresponding transition is in `exe` state. The column $f(other)$ shows the number of write tokens that other reference r_i^l may use at that time. The approach ensures to restore all the consumed tokens back to the bank, when the method ends its execution.

Table 3.4: The token distribution model for the safe (concurrent) execution of methods.

f(this)	**Semantics**	**f(bank)**	**f(bank′)**	**f(other)**
$0<fr_i^j<1 \wedge fw_i^j=1$	Unique	$0<fr_i^B<1, fw_i^B=1$	$0<fr_i^B<1, fw_i^B=0$	$\forall l\neq j: fw_i^l=0$
$0<fr_i^j<1 \wedge fw_i^j=1$	Share	$0<fr_i^B<1, fw_i^B=1$	$0<fr_i^B<1, fw_i^B=0$	$\forall l\neq j: fw_i^l=0$
$0<fr_i^j<1 \wedge fw_i^j=1$	Full	$0<fr_i^B<1, fw_i^B=1$	$0<fr_i^B<1, fw_i^B=0$	$\forall l\neq j: fw_i^l=0$
$0<fr_i^j<1 \wedge fw_i^j=0$	Immutable	$0<fr_i^B<1, 0<fw_i^B\leq 1$	$0<fr_i^B<1, 0<fw_i^B<1$	$\forall l\neq j: fw_i^l<1$
$0<fr_i^j<1 \wedge fw_i^j=0$	Pure	$0<fr_i^B<1, 0<fw_i^B\leq 1$	$0<fr_i^B<1, 0<fw_i^B<1$	$\forall l\neq j: fw_i^l<1$

- It generates the fully discrete model of the updated semantics model (Equation 3.5). For which, it modifies the token distribution model given in Equation 2.5, for `full` and `share` permission which shows that the methods requiring read-write access on the referenced object would always consume all of the available (write) tokens in the bank.

$$N_w : AP \quad \rightarrow \quad \{0, 1, \ldots, K+1\}, \quad N_w(a) = \begin{cases} 0, & \text{if a} \in \{\bot, \text{Immutable, Pure}\} \\ K+1, & \text{if a} \in \{\text{Unique, Full ,Share}\} \end{cases} \tag{3.5}$$

2. **Model Generation.**
   It then encodes the generated semantics into an abstract state-machine model, acceptable by `evmdd-smc` model checker in Pulse. For which, it generates (redefines) state transition rules, for starting and ending the execution of a method, with `immutable`, `share`, `pure` and `full` permission. The objective was to enable the model-checker to consider all possible side effects and to automatically perform the comprehensive concurrency analysis of the underlying program, based on the five types of inferred permissions.
   For example, a transition where the reference r_i^j starts a method m_i^k with `pure` or `immutable` permission would be defined as.
   - The **guard expression** first ensures that the reference r_i^j exists and $ap_i^j \neq \bot \wedge pc_i^j = (done,.) \wedge state_i = t_i^h \wedge tkr_i^B \leq 1 \wedge tkw_i^B > 0$
   - The **update expression** is $pc_{i\prime}^j = (exe, .) \wedge method_{i\prime}^j = k \wedge ap_{i\prime}^j = 5 \wedge tkr_{i\prime}^B = tkr_i^B - 1 \wedge tkw_{i\prime}^B = tkw_i^B - 1 \wedge tkr_{i\prime}^j = tkr_i^j + 1 \wedge tkw_{i\prime}^j = tkw_i^j + 0$

   A transition where the reference r_i^j ends a non-constructor method m_i^k with `pure` or `immutable` permission would be then.
   - The **guard expression** ensures that r_i^j exists and it is currently executing the method m_i^k. The guard is $pc_i^j = (exe,.) \wedge method_{i\prime}^j = k$
   - The **update expression** is $pc_{i\prime}^j = (done, .) \wedge tkr_{i\prime}^B = tkr_i^B + 1 \wedge tkw_{i\prime}^B = tkw_i^B + 1 \wedge tkr_{i\prime}^j = tkr_i^j - 1 \wedge tkw_{i\prime}^j = tkw_i^j - 0$

   The guard expression, for a starting a method with `immutable` permission checks whether there are enough read tokens in the bank to fulfill the required permission of the current reference r_i^j. Further, it checks the number of write tokens in the bank to ensure that no other reference, to the same object, is currently executing a method with `write` permission.

The objective is to enforce the mutual exclusion mechanism between two methods accessing the same object where at least one of them is writing on it.

3. **Method-pair Concurrency Analysis.** Finally, it parses the results of the model-checker and computes the number (percentage) of method pairs that can be parallelised with each other (comparing a method with itself e.g., `concurrent(m1, m1)`) over a state-space of (total pairs) methods in the given program. For each class, it calculates the number of concurrent (method) pairs following the binomial coefficient formula given below, choosing two (02) methods from `n` methods. Further, it incorporates the results of extended (concurrency) analysis in the Pulse generated report as explained in Section 4.3.

$$\binom{n}{2} + n = \frac{n!}{k!(n-k)!} + n$$

where `n` is the total number of methods having permission contracts.
Table 3.5 shows an adjacency matrix showing methods (pair) concurrency analysis after extending the Pulse analysis. All the blue ✓ symbols show the concurrency analysis that the Pulse tool performs based on the permission compatibility and side effects analysis, and all the red ✓ symbols show the extended method pairs concurrency analysis of the underlying program based on the permission-based side effect analysis of the methods.

Table 3.5: Extended method pair concurrency analysis.

		APs(m2)				
		unique	full	share	immutable	pure
APs(m1)	unique	✓	✓	✓	✓	✓
	full	✓	✓	✓	✓	✓
	share	✓	✓	✓	✓	✓
	immutable	✓	✓	✓	✓	✓
	pure	✓	✓	✓	✓	✓

The implementation details of the permission checking mechanism and the results of the extended concurrency analysis, for the example program given in Listing 1.1, are presented in Chapter 4.3.

Chapter 4

Implementation

The goal of this chapter is to describe the detailed design and implementation of the Sip4J tool along with its integration with the permission-based model checking tool Pulse. The implementation is based on the permission inference and checking approach presented in Chapter 3. The core functionalities of the proposed approach are implemented as a Java Eclipse plugin and it is a freely available tool[1]. The `Sip4J` framework not only complements the existing permission-based verification and parallelization approaches such as Plural (Bierhoff, 2006; Bierhoff and Aldrich, 2007, 2008), Pulse (Cataño et al., 2014) , Plaid (Aldrich et al., 2012), and Æminium (Stork et al., 2014) to perform their intended tasks without posing extra work on programmers but can also be used to employ them for the general-purpose program development and verification purpose.

This chapter provides a high-level system architecture of the tool and an overview of the underlying approach. It first explains the requirements to integrate the Pulse tool as a part of permission inference framework. Further, it elaborates the artifacts of permission inference and checking mechanisms, by revisiting the motivating example given in Listing 1.1. Finally, it presents the implications and limitations of the `Sip4J` framework and its analysis in general.

4.1 System Architecture and Overview

The high level system architecture of the `Sip4J` framework mainly consists of two modules: (1) Permission Extractor and (2) Permission Checker as shown in Figure 4.1. The `Sip4J` framework takes an un-annotated sequential Java program as input. It produces an annotated version of the input program with access permission contracts defined at the method level. It integrates Pulse, a permission-based model checking tool to automatically verify the correctness of the inferred specifications and to reason about their concurrent behavior.

The framework produces the following artifacts:

- Five types of symbolic permissions, following the permission semantics 2.1, generated at the object's (class) field level thereby, generating permissions at a more granular level that can be used to exploit the maximum concurrency present in the code.
- A Plural annotated version of the input Java program, following the Plural specifications 2.2.1, as acceptable by the Pulse tool.

[1] https://github.com/Sip4J/Sip4J

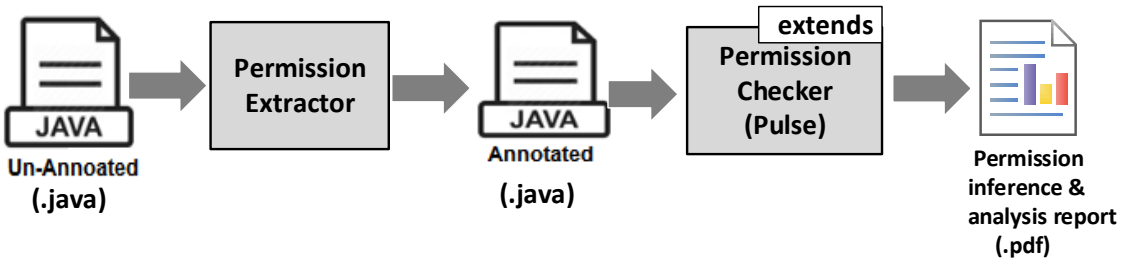

Figure 4.1: A high-level work-flow depiction of the Sip4J framework

- An analysis report (in PDF with visualizations) comprising the correctness, concurrency and, code reachability analysis of the inferred specifications through Pulse 2.2.2, along with the Plural annotated version of the input Java program.

The permission extractor and checker are explained in more detail in Section 4.2 and Section 4.3 respectively.

4.2 Permission Extractor

The permission extractor module in `Sip4J` reveals the permission-based implicit dependencies present between the code (methods) and the global states in an un-annotated Java program in the form of high-level abstractions i.e., symbolic permissions. The final output is the annotated version of the input program (a .java file) with the Plural annotated version of the input program where access permission contracts are generated on the receiver object (`this`), using a single typestate i.e., '`alive`', and following the Design by Contract principle acceptable by the permission checker module.

The permission extractor consists of four modules: 1) Metadata Extractor; 2) Graph Constructor; 3) Access Permissions Generator; and 4) Plural Annotations Generator as shown in Figure 4.2.

Phase 1

The metadata extractor performs modular static analysis (data flow, alias flow, and context analysis) of the input program based on its Abstract Syntax Tree (AST). The analysis generates and traverses the AST of the source code using the AST visitor. For each method in the input program, it parses the method's signature and body, using the AST parser, to extract (maintain) the object's access as read, write and aliasing information in the method.

The analysis is based on the type of expressions encountered in an expression statement such as `<FIELD_ACCESS>`, `<ASSIGNMENT>`, etc., and the type of reference variables accessed in each expression (class fields, method's local variables or parameters). The approach maps the method's local references and parameters with their global reference (alias) if any. This information is then used to extract the read, write and aliasing information of the referenced object against them, to maintain the integrity of data during parsing. The approach ignores all the local references (and parameters) that are not aliases of any global references. This is because manipulating local objects in a method does not affect the access rights of the

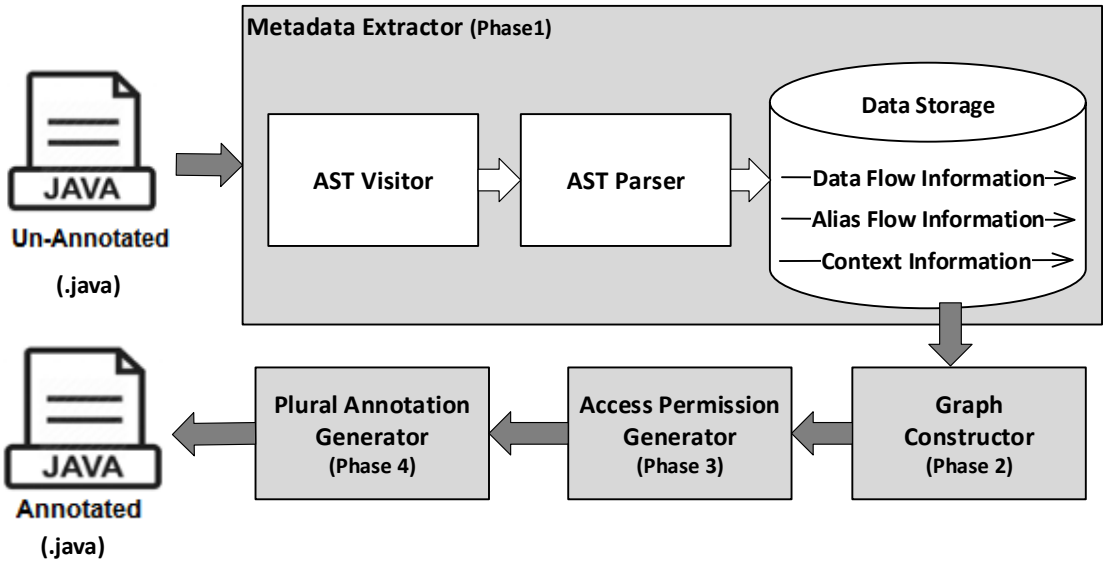

Figure 4.2: The working process of the permission extractor module in Sip4J framework.

current and other methods. The approach performs flow-insensitive analysis of the source code ignoring the order of execution of statements. It preserves the semantics of assignment statements by determining the type of a reference variable on the left-hand side of an assignment statement based on its right-hand side expression. This information is then used to precisely extract the data-flow and alias-flow information of the referenced objects during parsing. Further, the context analysis (access by the other methods) of an object accessed in the current method is based on its data-flow and alias-flow analysis across other methods.

The output of this module is a data structure maintaining the data-flow, alias-flow and context information for all the objects shared at the method level.

Phase 2

The graph constructor generates a permission-based graph model for each method, based on the metadata extracted in phase 1. It first maps the extracted information (data-flow, alias-flow and context information) in the form of pre-defined graph notations. The object's accesses are then modelled as read, write and alias edges between the variable (object) and method nodes.

Phase 3

The permission generator module traverses the permission graph constructed for each method in phase 2, by following the read, write and alias edges between the variable and method nodes. The graph traversal is simple and computationally fast as it does not involve any cycles and expensive steps like backtracking. The output is five types of symbolic permissions such as `unique`, `full`, as pre- and post-permission, for the object's (class) fields accessed in a method. It also generates a special `none` permission to represent the absence of permission. This happens when a method creates a `null reference` or when a method does not access

a shared object.

4.2.1 Motivating Example Revisited: The Access Permission Contracts

Listing 4.1 shows the output of phase 3 for the un-annotated example program, given in Listing 1.1, with permission contracts defined at the field level following the Plural syntax (Section 2.2.1) without typestate information. As discussed previously in Section 2.2.2, the approach adds the typestate (`alive`) information to perform the evaluation of the inferred specifications by the Pulse tool, otherwise, inferring typestate is not objective of the research in this thesis.

In Listing 4.1, we refer parameters by using identifiers of the actual objects against them. This is because the underlying approach maps the local references and formal parameters with their global references (aliases), to extract the data-flow and aliasing information of the actual objects, and to maintain the integrity of the data during parsing. For example, in Listing 1.1 in Line 12, the parameter `coll` in method `incrColl()` is mapped with its actual reference `array2` against the method call, `incrColl(obj1.array2)` given in Listing 1.1 at Line 68. The implication of generating access permission contracts on at the object's field level is discussed in Section 4.4.

Listing 4.1: Access permission contracts for the example program given in Listing 1.1

```
1 class ArrayCollection{
2  public  Integer[] array1 = new Integer[10];
3  @Perm(ensures="unique(array1)")
4  public ArrayCollection() {}

6  @Perm(requires="pure(array1)",
7        ensures="pure(array1)")
8  public void printColl(Integer[] coll) {}

10 @Perm(requires="share(array1) * share(array2)",
11       ensures="share(array1) * share(array2)")
12 public void incrColl(Integer[] coll) {}

14 @Perm(requires="pure(array1)",
15       ensures="pure(array1)")
16 public boolean isSorted(Integer[] coll) {}

18 @Perm(requires="pure(array1)",
19       ensures="pure(array1)")
20 public Integer findMax(Integer[] coll) {}

22 @Perm(requires="pure(array1)",
23       ensures="pure(array1)")
24 public void computeStat(Integer[] coll){}

26 @Perm(requires="unique(array1) * unique(array2)",
27       ensures="none(array1)* none(array2)")
28 public void tidyupColl(Integer[] coll){}
29 }
30 ENDOFCLASS
31 class ObjectClass{
32 public Integer[] array2 = new Integer[10];
33 public  Client x = new Client(), y = new Client();
34 public  Client z = new Client(),w = new Client();
35 @Perm(requires="full(x) * full(y) * full(w) * immutable(z)",
36       ensures ="full(x) * full(y) * full(w) * immutable(z) ")
37 public void manipulateObjects(Client p1, Client p2){}
38 }
39 ENDOFCLASS
40 class Client{
41 Integer data = 100;
42 @Perm(requires="none(obj1) * none(obj2)",
43       ensures="unique(obj1) * unique(obj2) ")
44 public static void main(String[] a) {
45  ArrayCollection obj1 = new ArrayCollection();
```

```
46   ObjectClass obj2 = new ObjectClass();
47   obj1.incrColl(obj2.array2);
48   obj1.computeStat(obj1.array1);
49   obj1.computeStat(obj2.array2);
50   obj1.tidyupColls(obj2.array2);
51   obj2.manipulateObjects(obj2.w,obj2.z);
52 }
53 }
54 ENDOFCLASS
```

Consider the permission contract for method `printColl()` in Listing 4.1 at Line 6 & 7, "`@Perm(requires="pure(array1)", ensures="pure(array1)")`". The contract is generated following the method call expression `obj1.computeStat(obj1.array1)` in Listing 1.1 at Line 69. It states that the method needs `pure` permission, as pre-permission, on the object referenced by variable `array1`. The post-permission specifies that the method guarantees to return the consumed permissions, on the same object, to the caller of the method.

Phase 4

In phase 4, the Plural annotation generator produces a Plural annotated version of the input Java program i.e., a Java program with access permission contracts and typestate information. For this purpose, it encodes the specifications generated in phase 3 to the Plural specifications, where permissions are defined on the receiver object (using the keyword "`this`") with a single typestate `alive`, acceptable by the permission checking module in Pulse.

To reconcile the differences between the phase 3 output and the Pulse acceptable program, a number of characteristics of Pulse and Plural are worth noting.

Pulse Integration Requirements

- Firstly, Plural is a permission-based specification language where permission contracts are defined at the object level using the keyword "`this`". Pulse tool does not support permission annotations at the field level e.g. `pure(this.field)` and `pure(obj.field)` etc. However, the `Sip4J` tool generates permissions at the field level that can help enable concurrency at a more granular level.
- Secondly, the Pulse tool does not support `share` and `immutable` permission as a part of its analysis. This is because, referring to the permission coexistence semantics in Table 2.1, a `share` permission cannot coexist with either `unique`, `full` and `immutable` permission on the same object. Similarly, `immutable` permission cannot coexist with either `share` or `full` permission. Hence, Pulse's does not support share and immutable as a part of its concurrency analysis.
- Thirdly, Plural follows the Design by Contract principle to specify permission contracts, as pre- (P) and post- permissions (Q), where the relation $P \stackrel{!}{=} Q$ should hold i.e., the pre- and post-conditions should be the same, to maintain the integrity of the specifications and to perform program verification based on these specifications.
  However, the situation can be different in reality. For example, in Listing 4.1 in Line 26 & 27, the permission extractor (phase3) generates different pre- and post-permission for method `tidyupColl()`. This is because, in Listing 1.1 in Line 43, the method creates a

null reference using the reference variable `array1`. In this case, the permission extractor generates a special permission i.e. `none`, as post-permissions on `array1`, to represent that new instance of the array object should be created with `unique` permission, for other methods to access it without generating the `null-pointer` exception in the program.

- Fourthly, Pulse always requires a non-parameterized (default) constructor with `unique` permission on the receiver object (`this`) to perform the permission correctness and concurrency analysis of the non-constructor methods accessing the same object.

Generating Plural Annotations: The permission checker, in phase 4, generates the Pulse translated version (Plural specifications) of the input Java program in the following ways.

- It always defines a non-parameterized (default) constructor, even if no explicit constructor is defined in the class, to generate `unique` permission on the receiver object (`this`) (see Listing 4.2, line 4).
- As mentioned previously in Section 3.3.2, Pulse does not support overloaded methods (including constructors) as a part of its correctness and concurrency analysis. We need to ignore the overloaded methods in the Plural annotated version of the input program.
- For all the non-constructor methods, the Plural annotation generator produces the conservative or safe permissions on the receiver object i.e. `(this)` using the access permissions contracts generated at the field level in phase 3. For this purpose, it maps the five types of symbolic permissions on a scale 1 to 5 where value 1 represents the most relaxed (`immutable`) permission and value 5 shows the most conservative i.e, `unique` permission. It then computes the maximum of the pre- and post- permissions generated at the object's field level in phase3, to generate pre- and post-permissions of the associated object (`this`). For example, for method `manipulateObject()`, in Listing 4.2 in Line 32 & 33, it generates `full` permission on the receiver object (`this`), as a safe option, although in Listing 4.1 at Line 35 & 36, the required permission on some of the fields such as `z` is `immutable`.
- For non-constructor methods, it generates notation `<AP>(#i)` to represent pre- and post-permissions for the parameters. For example, the parameter notations `full(#0)` and `pure(#1)` (see Listing 4.2 in Line 32 & 33) are replacement of the actual references (`full(w)` and `pure(z)`), given in Listing 4.1 in Line 35 & 36.
- For non-constructor methods, it generates permission contracts following the relation $P \stackrel{!}{=} Q$ where pre- (P) and post-conditions (Q) should be the same (see Listing 4.2, line 6 & 7) However, in reality this relation could be different, as explained in the previous section using the access permission contract for method `tidyupColl()` in Listing 4.1 at Line 27.
- It automatically adds other required annotations as a part of the Plural annotated version of the Java program such as:
  - An import statement to support Plural annotations (Listing 4.2, line 1) as a part of the Java program.
  - It adds typestate '`alive`' using `@States` statement at the class level (Listing 4.2 in

Line 2).

– Typestate as a part of pre- and post- permission for the referenced object (Listing 4.2 in Line 6 and 7).

– The annotation `ENDOFCLASS` at the end of each class (Listing 4.2 in Line 27, 36 & 45).

4.2.2 Motivating Example Revisited: The Plural Annotations

Listing 4.2: A Plural annotated version of the Java program given in Listing 4.1

```
1 import edu.cmu.cs.plural.annot.*;
2 @ClassStates({@State(name = "alive")})
3 class ArrayCollection{
4 @Perm(ensures="unique(this) in alive")
5 ArrayCollection() {   }
6 @Perm(requires="pure(this) in alive * pure(#0) in alive",
7 ensures="pure(this) in alive * pure(#0) in alive")
8 public void printColl(Integer[] coll) {
9 }
10 @Perm(requires="pure(this) in alive * pure(#0) in alive",
11       ensures="pure(this) in alive * pure(#0) in alive")
12 public void computeStat(Integer[] coll) {
13 }
14 @Perm(requires="pure(this) in alive * pure(#0) in alive",
15      ensures="pure(this) in alive * pure(#0) in alive")
16  boolean isSorted(Integer[] coll) { }
17 @Perm(requires="pure(this) in alive * pure(#0) in alive",
18      ensures="pure(this) in alive * pure(#0) in alive")
19  Integer findMax(Integer[] coll) { }
20 @Perm(requires="share(this) in alive * share(#0) in alive",
21       ensures="share(this) in alive * share(#0) in alive")
22 public void incrColl(Integer[] coll) { }
23 @Perm(requires="unique(this) in alive * unique(#0) in alive",
24 ensures="unique(this) in alive * unique(#0) in alive")
25 public void tidyupColls(Integer[] coll) { }
26 }
27 ENDOFCLASS
28 @ClassStates({@State(name = "alive")})
29 class ObjectClass {
30 @Perm(ensures="unique(this) in alive")
31 ObjectClass() {   }
32 @Perm(requires="full(this) in alive * full(#0) in alive * pure(#1) in alive",
33     ensures="full(this) in alive * full(#0) in alive * pure(#1) in alive")
34 void manipulateObjects(Client p1, Client p2) { }
35 }
36 ENDOFCLASS
37 @ClassStates({@State(name = "alive")})
38 class Client {
39 @Perm(ensures="unique(this) in alive")
40 Client() {   }
41 @Perm(requires="unique(this) in alive",
42       ensures="unique(this) in alive")
43 public static void main(String[] args) { }
44 }
45 ENDOFCLASS
```

The annotations defined for the Pulse translated version in Listing 4.2 also shows the minimum annotation overhead imposed by the existing permission-based verification approaches such as Plural and Pulse, to verify program behavior. Therefore, `Sip4J` framework, by inferring access permission contracts from the source code of an input program, can help existing permission-based approaches to perform their intended tasks, without posing extra work on programmers.

4.3 The Permission Checker

As the second module of the `Sip4J` framework, the permission checker aims at verifying the correctness and effectiveness of the inferred specifications. For this purpose, it integrates and extends Pulse, a permission-based model checking tool implemented as a Java Eclipse plug-in, to automatically verify the correctness of the inferred specifications (i.e., annotation in Java code). As discussed previously in Section 2.2.2, Pulse uses `evmdd-smc` model-checker (Roux and Siminiceanu, 2010) to identify errors in the input specifications and to reason about the program behavior in terms of its concurrency and code reachability analysis.

Figure 4.3 illustrates the working process of the permission checker module, where blue rectangles with the dotted boundary show the Pulse modules (Model Generator and Parser), modified by our approach to extend the Pulse concurrency analysis and to incorporate the results of the analysis in the generated report.

Before elaborating the permission checker mechanism in the Pulse tool, we revisit the extensions made by the `Sip4J` framework as a part of the Pulse concurrency analysis.

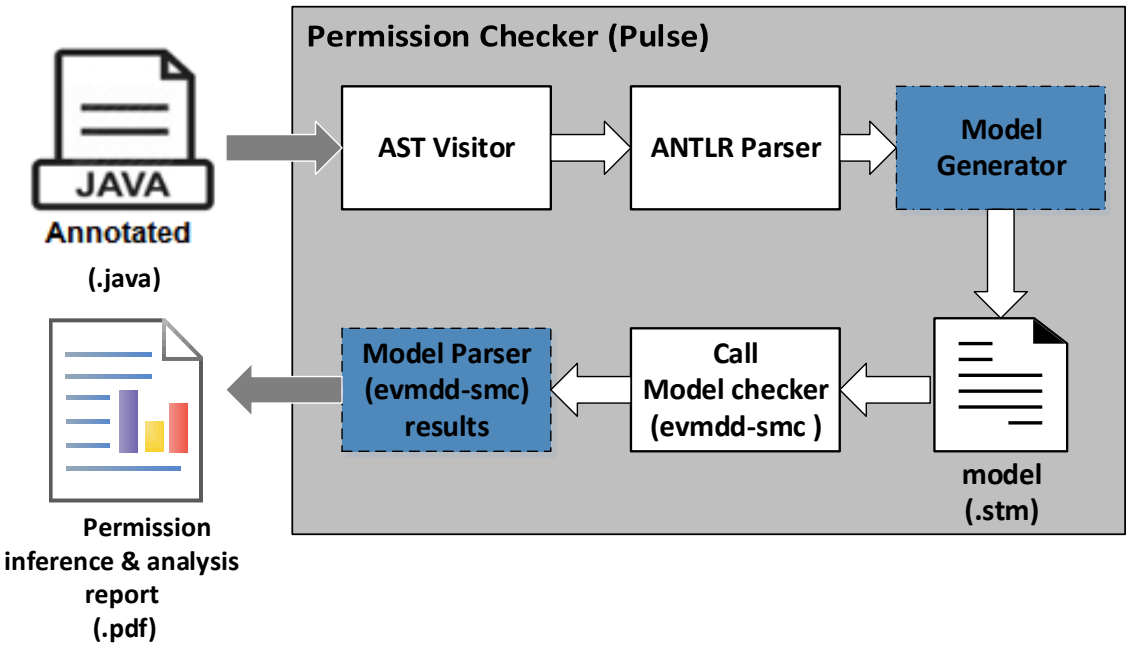

Figure 4.3: The working process of the permission checker module in Sip4J framework.

A Revisit of the Extensions made in the Pulse tool

The permission checker extends the permission checking mechanism in the Pulse tool in following two ways.

Concurrency Analysis. As explained previously, the concurrency analysis in the Pulse tool is limited and it does not support `share` and `immutable` permissions as a part of its concurrency analysis. However, this information is necessary to explicitly show the method's side effects in the program, and to avoid the data races of the form <read-write> and `<write-write>`, that can be created by the methods having `share` and `immutable` permissions as a part of their permission contracts.

For this purpose, the permission checker modifies the **Model Generator** in the Pulse tool. It first generates the discrete state semantics, of the input specifications, for `share`, `immutable`, `pure` and `full` permissions and encodes them into a state machine model, in `model.stm` file, as acceptable by the `evmdd-smc` model checker. Further, it parses the output of the model-checker, through the **Model Parser** module, to compute the number of concurrent method pairs, based on the extended permission model.
The theoretical background of the extended concurrency analysis is already explained in Section 3.3.2 which follows the permission encoding and model generation mechanism in Pulse (Section 2.2.2).

Report Generation. The `Sip4J` framework extends the Pulse analysis report in two ways a) by automatically adding the Plural annotated Java program and, b) by incorporating the results of method pair concurrency analysis, for the input program, in the analysis report. The complete analysis report for the example and other benchmark programs can be found in the `Sip4J` project repository on GitHub[2]. The objective was to provide readers a quick, explicit and an integrated view of the program with the inferred permissions (method-level dependencies) along with its correctness, concurrency and code reachability analysis at one place.

Now, we will elaborate on different phases of the permission checker module in the `Sip4J` framework.

The permission checker, as the first step, uses the AST visitor to traverse input Java program with annotations. It then parses the input program and feeds the relevant information (annotations) to the `ANTLR` (ANother Tool for Language Recognition) parser. The `ANTLR` parser uses a pre-defined grammar to parse the input specifications. It checks the misspelled specifications (if any) in the input program.

The next step feeds the output of the ANTLR parser into the model generator module in Pulse as input. The model generator then encodes the input specifications into (semantically equivalent) discrete state semantics model and saves the translation results in a `model.stm` (file), acceptable by the symbolic `evmdd-smc` model checker (Roux and Siminiceanu, 2010).

The evmdd-smc model-checker is inspired by the SAL (Symbolic Analysis Laboratory)(L. de Moura, 2003), its input language is similar to SAL; however, evmdd-smc is more efficient than SAL for several reasons. Firstly, evmdd-smc is powered by an edge-valued decision diagram (EVMDD) library, libevmdd[3] that can be orders of magnitude faster than the ubiquitous CUDD, especially for models that capture concurrency. Secondly, evmdd-smc is free of the syntactic sugar provided by the SAL, which often poses tremendous pre-processing overhead. The use of CTL formulae gives model-checker further freedom to perform verification tasks tailored to each application. The `evmdd-smc` model checker analyzes the core integrity properties of the underlying program using the pre-defined CTL formulae generated as the part of model in `model.stm` file.

There are four sections of the generated model in `model.stm` file, namely, Declarations,

[2] https://github.com/Sip4J/Sip4J
[3] http://research.nianet.org/radu/evmdd/.

Initial States, Transitions and CTL Properties (the details of each section can be found in Section 2.2.2). To generate the truly concurrent settings in the model, the value of K is set as $k=4$, that creates four co-existing references (aliases) of a referenced object in the generated model.

For the sake of brevity, we present here all four parts of the generated model, for some of the methods of the example program given in Listing in 4.2. The complete model (`model.stm` file) of the example program, given in Listing 4.2, can be found in the `Sip4J` project repository as mentioned above.

Figure 4.4 and Figure 4.5 show the variable declaration and initialization part of the generated model for class `ArrayCollection`, with access permissions mapped in the range [0, 5] and initialized as required. The code for the `start` and `end` of the transitions, for method `ArrayCollection()` and `incrColl()` along with their guard and update formulae, is shown in Figure 4.6 and 4.7 respectively. Figure 4.8 shows the CTL properties generated for method `ArrayCollection()` and `incrColl()` while the concurrency analysis of method pairs (`ArrayCollection(), incrColl()`) and (`incrColl(), printColl()` is shown in Figure 4.9.

```
Declarations
/*ArrayCollection-->*/
   state_ArrayCollection_0        [0, 2] /* typestate information */
   tkrB_ArrayCollection_0         [0, 5] /* 5 read token in bank */
   tkwB_ArrayCollection_0         [0, 5] /* 5 write tokens in bank */\\

   pc_ArrayCollection_0_0         [0, 1] /* program counter to execute method for alias k = 0 */
   method_ArrayCollection_0_0     [0, 7] /* 7 class methods excluding constructor, k = 0 */
   ap_ArrayCollection_0_0         [0, 5] /* 5 types of access permission to execute method, k = 0  */
   tkr_ArrayCollection_0_0        [0, 5] /* 5 read tokens for alias k = 0 */
   tkw_ArrayCollection_0_0        [0, 5] /* 5 write tokens for alias k = 0 */

   pc_ArrayCollection_0_1         [0, 1] /* program counter to execute method for alias k = 1 */
   method_ArrayCollection_0_1     [0, 7] /* 7 class methods excluding constructor, k = 1 */
   ap_ArrayCollection_0_1         [0, 5] /* 5 types of access permission to execute method, k = 1 */
   tkr_ArrayCollection_0_1        [0, 5] /* 5 read tokens for alias k = 1 */
   tkw_ArrayCollection_0_1        [0, 5] /* 5 write tokens for alias k = 1 */

/*<--ArrayCollection*/
```

Figure 4.4: Variable declarations for ArrayCollection class

At the next step, the model checker is invoked on the `model.stm` file, to verify the input specifications based on the generated transitions, and the pre-defined CTL properties for each method. The model checker generates an abstract state machine model of the annotated program and performs a reachability graph analysis of the generated state-space. It identifies the missing specifications and ensures that access permission contracts do not violate their intended semantics, using the method `requires` clause satisfiability analysis. Moreover, it identifies the concurrent method pairs by considering the method's side effects based on the input specifications.

Finally, the permission checker module parses the output of the model checker analysis and generates a user-friendly (.pdf) report of the program comprising the correctness, concurrency and code reachability analysis of the input specifications, along with the Plural annotated version of the input program.

```
95  Initial states
96  /*ArrayCollection-->*/
97      state_ArrayCollection_0        =0  /* undefined */
98      tkrB_ArrayCollection_0         =5  /* initial read tokens in bank */
99      tkwB_ArrayCollection_0         =5  /* initial write tokens in bank */
100
101     pc_ArrayCollection_0_0         =1 /* program counter is set to 1 for alias k = 0  */
102     method_ArrayCollection_0_0     =0 /* undefined */
103     ap_ArrayCollection_0_0         =0 /* undefined */
104     tkr_ArrayCollection_0_0        =0 /* undefined */
105     tkw_ArrayCollection_0_0        =0 /* undefined */
106
107     pc_ArrayCollection_0_1         =1 /* program counter is set to  1 for alias k = 1 */
108     method_ArrayCollection_0_1     =0 /* undefined */
109     ap_ArrayCollection_0_1         =0 /* undefined */
110     tkr_ArrayCollection_0_1        =0 /* undefined */
111     tkw_ArrayCollection_0_1        =0 /* undefined */
112
113 /*<--ArrayCollection*/
```

Figure 4.5: Variable initializations for ArrayCollection class

```
Transitions
/*K=0-->*/
/*ArrayCollection-->*/

 start_ArrayCollection_ArrayCollection_0_0: /* start transition for the constructor method */
   ap_ArrayCollection_0_0 = 0  /\ /* program counter for the current method for k = 0 */
   ap_ArrayCollection_0_1 = 0  /\ /* program counter for k = 1, no other method is in executing */
   ap_ArrayCollection_0_2 = 0  /\ /* program counter for k = 2, no other method is in executing */
   ap_ArrayCollection_0_3 = 0  /\ /* program counter for k = 3, no other method is in executing */
   ap_ArrayCollection_0_4 = 0     /* program counter for k = 4, no other method is in executing */
    ->
   pc_ArrayCollection_0_0' = 0  /\     /* method is in executing state */
   method_ArrayCollection_0_0' = 1 /\  /* method number 1 */
   ap_ArrayCollection_0_0' = 1   /\    /* unique */
   tkr_ArrayCollection_0_0' = 5 /\ tkw_ArrayCollection_0_0' = 5 /* all read and write tokens gone */
   tkrB_ArrayCollection_0' = 0  /\ tkwB_ArrayCollection_0' = 0 /* bank read and write tokens empty */

 end_ArrayCollection_ArrayCollection_0_0: /* end transition for the constructor method  */
   pc_ArrayCollection_0_0 = 0  /\
   method_ArrayCollection_0_0 = 1
  ->
   pc_ArrayCollection_0_0' = 1  /\ /* method in done state */
   state_ArrayCollection_0' = 1 /\ /* state transition to alive state */
   tkrB_ArrayCollection_0' = 5  /\ tkwB_ArrayCollection_0' = 5  /* read and write tokens deposit back to bank */
   tkr_ArrayCollection_0_0' = 0 /\ tkw_ArrayCollection_0_0' = 0 /* all the read and write tokens returned */

/*<--ArrayCollection*/
```

Figure 4.6: The start and end transition for ArrayCollection() method

Figure 4.10 shows a screenshot of the `Sip4J` tool for the example program which is given in Listing 1.1 along with its analysis report.

4.3.1 Motivating Example Revisited: Pulse Analysis

This section presents the results of the correctness and concurrency analysis, performed by the `Sip4J` framework, for the Plural annotated version of the example program given in Listing 4.2.

Correctness Analysis. Figure 4.11 shows the results of the satisfiability of pre-conditions (the `requires` clause) for all the methods in Listing 4.2. The analysis shows that all the methods got their required (safe) permissions to start their execution. In other words, there is no unreachable method and no specification errors in the input program that can otherwise

```
/*<--ArrayCollection*/

 start_ArrayCollection_incrColl_0_0:
    pc_ArrayCollection_0_0  = 1 /\  /* done */
    ap_ArrayCollection_0_0 != 0 /\  /* access permissions should not be undefined*/
    state_ArrayCollection_0 > 0 /\  /* state is alive */
    tkrB_ArrayCollection_0 > 0  /\  /* read token in bank must be greater than 0 */
    tkwB_ArrayCollection_0 = 5      /* write tokens must be the total count = 5 */
    ->
    pc_ArrayCollection_0_0' = 0 /\  /* executing */
    method_ArrayCollection_0_0' = 6 /\  /* method number 6 = incrColl()*/
    ap_ArrayCollection_0_0' = 3   /\   /* share permission is defined */
    tkrB_ArrayCollection_0' = tkrB_ArrayCollection_0 - 1 /\ /* some read tokens taken */
    tkwB_ArrayCollection_0' = 0  /\  /* all write tokens taken */
    tkr_ArrayCollection_0_0' = 1 /\  /* give 1 read token to alias k = 0 */
    tkw_ArrayCollection_0_0' = 5     /* give all write tokens to alias k = 0 */

 end_ArrayCollection_incrColl_0_0:
    pc_ArrayCollection_0_0 = 0  /\  /* executing */
    method_ArrayCollection_0_0 = 6  /* method number 6 = incrColl() */
    ->
    pc_ArrayCollection_0_0' = 1  /\   /* done */
    state_ArrayCollection_0' = 1   /\ /* alive */
    tkrB_ArrayCollection_0' = tkrB_ArrayCollection_0 + 1 /\  /* The read token is deposit to bank */
    tkwB_ArrayCollection_0' = 5  /\  /* all the write tokens deposit to bank */
    tkr_ArrayCollection_0_0' = 0 /\  /* all the read tokens returned */
    tkw_ArrayCollection_0_0' = 0 /\  /* all the write tokens returned */

 /*<--ArrayCollection*/
```

Figure 4.7: The start and end transition for incrColl() method

```
/* checks method satisfiability (reachability) */
requires_clause_of_ArrayCollection_ArrayCollection_0_0:
   method_ArrayCollection_0_0 = 1    /\ pc_ArrayCollection_0_0 = 0 /* requires clause for method ArrayColl()*

requires_clause_of_ArrayCollection_incrColl_0_0: /*require clause for method incrColl()*/
  method_ArrayCollection_0_0 = 6    /\ pc_ArrayCollection_0_0 = 0

/* checks state tranistion for the referenced object */

stateTransition_of_ArrayCollection_from_alive_to_alive:
    state_ArrayCollection_0 = 1    /\ EX(state_ArrayCollection_0 = 1)
```

Figure 4.8: CTL model for the reachability (requires clause) analysis of ArrayCollection() and incrColl() methods

```
/* checks concurreny among methods accessing the same referenced object*/

concurrentMethods_ArrayCollection_0_0_ArrayCollection_and_ArrayCollection_0_1_incrColl:
  /* method 1 and 6 both are in executing state with two aliases of the object, hence can not be parallelized
          method_ArrayCollection_0_0 = 1 /* method number 1 for alias k = 0 */ /\
          pc_ArrayCollection_0_0 = 0  /* executing for alias k = 0 */  /\
          method_ArrayCollection_0_1 = 6 /* method number 6 for alias k = 1 */ /\
          pc_ArrayCollection_0_1 = 0 /* executing for alias k = 1*//

concurrentMethods_ArrayCollection_0_0_incrColl_and_ArrayCollection_0_1_printColl:
   /* method 6 and 2 both are in executing state with two aliases of the object, hence can not be parallelize
           method_ArrayCollection_0_0 = 6 /* method number 6 for alias k = 0 */ /\
           pc_ArrayCollection_0_0 = 0   /* executing for alias k = 0 */  /\
           method_ArrayCollection_0_1 = 2 /* method number 2 for alias k = 1 */ /\
           pc_ArrayCollection_0_1 = 0  /* executing for alias k = 1 */ /\

sinks    :    !EX(true) /* checks the sinkstates or deadlock if any*/
/*Arr
```


Figure 4.9: CTL model for the concurrency analysis of ArrayCollection(), incrColl() and printColl() methods

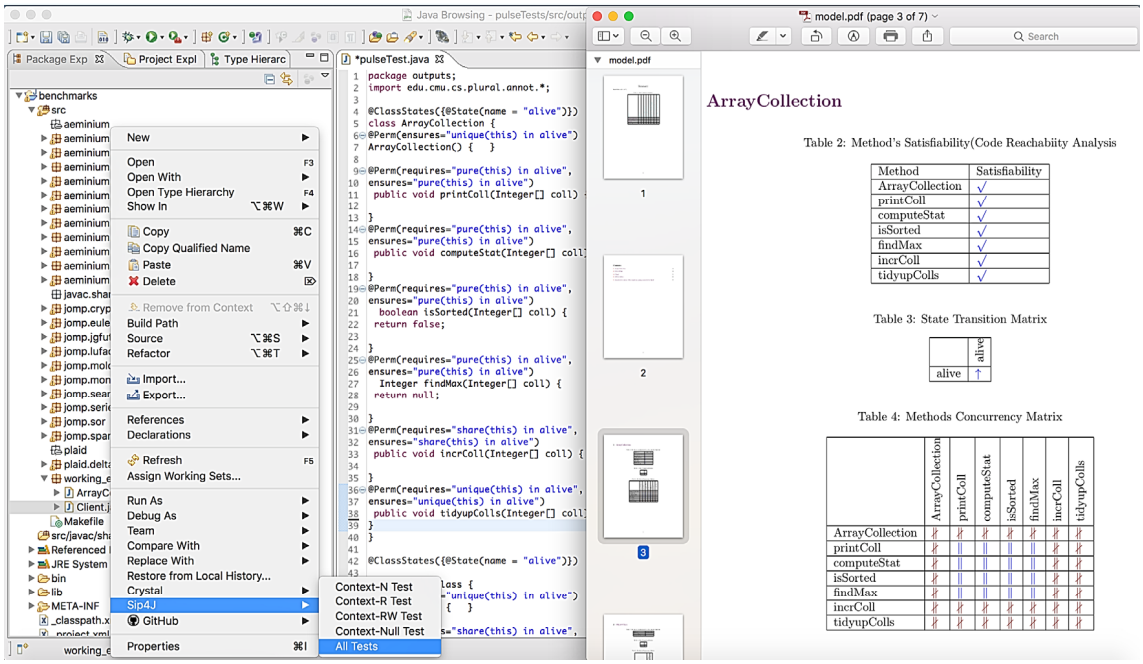

Figure 4.10: Sip4J screenshot, with its generated report for the ArrayCollection class given in Listing 1.1.

lead to unsatisfiability of the method contracts.

Method	Satisfiability
ArrayCollection	√
printColl	√
computeStat	√
isSorted	√
findMax	√
incrColl	√
tidyupColls	√

(a) Methods of class ArrayCollection.

Method	Satisfiability
ObjectClass	√
manipulateObjects	√

(b) Methods of class ObjectClass.

Method	Satisfiability
Client	√
main	√

(c) Methods of class Client.

Figure 4.11: Method satisfiability analysis of the example program

State Transition Analysis. In Figure 4.12, the symbol ↑ shows the possible state transition among typestates for all the methods in the example program. There is only one default typestate `alive` in the input specifications so the result will always be true. This is because all the typestates can transition to the root (`alive`) typestate. Alternatively, in the generated model,

there cannot be any deadlock due to the wrong or improper transition between typestates.

	alive
alive	↑

Figure 4.12: Typestate state transition matrix

Concurrency Analysis.
Figure 4.13 shows results of the concurrency analysis, generated by the Pulse tool itself and as extended by our approach to consider all possible side effects at the method level, for the example program given in Listing 4.2. The symbol ‖ in the concurrency matrix (Figure 4.13) indicates the possible parallel execution of the two methods, whereas the symbol ∦ shows the fact that the specified methods cannot be executed in parallel with each other.
For example, for the `ArrayCollection` class, the analysis shows that 4 out of 7 (57%) methods could potentially be executed, with at least one other method at the class level (Figure 4.13a). It shows that the constructor method cannot be parallelized with any other method. It further shows that methods that require `full or share` permission can not be run in parallel with other methods due to the side effects they can produce on each other when executed in parallel. Figure 4.13c shows a summary of the program with its extended concurrency analysis by `Sip4J` framework. For example, the concurrency (method pair) analysis of the `ArrayCollection` class shows that, in this class, 10 pairs (36%) of methods from a state space of total $\binom{7}{2}+7=28$ pairs can be executed in parallel.

4.4 Implications

We observe, that the `Sip4J` framework, by inferring access permission contracts not only helps programmers to work at a higher level of abstraction letting them focus on the functional and behavioural correctness of the program, but can also be used to automatically identify some of the syntactical errors in the program, such as *null-pointer* references, without actually compiling the program. Further, having contract-based specifications in an explicit way can help programmers to analyze the program behavior in terms of its code reachability analysis, without performing any code inspection.

We now revisit the motivating example, presented at the beginning of this section, to demonstrate the efficacy and expressiveness of permission-based specifications in verifying program behavior and enabling implicit concurrency present in a sequential program.
Null Pointers Analysis. With respect to the permission semantics, the null-pointer exceptions can arise in two ways in a program: (a) program error: reference to an object is a `null` reference itself and, (b) permission inference error: no method generates the permission required of a method, say `unique`, on the referenced object as its post-permission.
For example, in Listing 4.1, all the methods accessing the shared object, say `array1`, would cause a `null-pointer` exception and would remain unsatisfied (unreachable), if the client method, (in this case the constructor), does not generate the `unique` permission, by in-

	ArrayCollection	printColl	computeStat	isSorted	findMax	incrColl	tidyupColls
ArrayCollection	∦	∦	∦	∦	∦	∦	∦
printColl	∦	‖	‖	‖	‖	∦	∦
computeStat	∦	‖	‖	‖	‖	∦	∦
isSorted	∦	‖	‖	‖	‖	∦	∦
findMax	∦	‖	‖	‖	‖	∦	∦
incrColl	∦	∦	∦	∦	∦	∦	∦
tidyupColls	∦	∦	∦	∦	∦	∦	∦

(a) The ArrayCollection class.

	ObjectClass	manipulateObjects
ObjectClass	∦	∦
manipulateObjects	∦	∦

(b) The ObjectClass class.

Classes	Methods	States	Unreachable clauses	Unreachable states	Possible concurrent methods	Total. no. of method pairs	No. of concurrent method pairs	Percentage of concurrent methods pairs
ArrayCollection	7	1	0	0	4	28	10	36
ObjectClass	2	1	0	0	0	3	0	0
Client	2	1	0	0	0	3	0	0
Total Classes=3	11	3	0	0	4	34	10	29

(c) Summary of the extended concurrency analysis.

	Client	main
Client	∦	∦
main	∦	∦

(d) The Client class.

Figure 4.13: The extended concurrency analysis of the example program given in Listing 4.2

stantiating its object before using it. Similarly, the post-permissions (`none(array1)` and `none(coll)` of method `tidyupColl()` indicate that object `array1` and `array2` should be instantiated again, once the method `tidyupColl()` has been executed, for these objects to be used by other methods and without generating the `null-pointer` exception.
Further, the inference of access permission contracts can be used to define the way (order), these methods can be executed in parallel by considering their side-effects.

Method-level Dependencies Analysis. Furthermore, access permission contracts can be used to compute the dependencies between methods and to automatically impose ordering constraints.
For example, a close examination of the `requires` clause for method `tidyupColl()` in Listing 4.1 (Line 26), shows that the method can only be called if the objects `array1` and `array2` has `unique` permission as pre-permissions. This means it can either be called immediately after the constructor methods `ArrayCollection(array1)` and `ObjectClass(array2)`, as both generate (`unique`) permission on `array1` and `arry2`, or once all the methods accessing the `array1` and `array2` object have completed their execution, and `unique` permission on the referenced objects have been resumed, to be used by other methods.

Implicit Concurrency Analysis. Having such specifications (access permission contracts) in an explicit way, one can compute the number of immutable (independent) methods i.e., the methods that do not change the state of a shared object or the methods that should

always run sequentially. In this respect, access permission splitting and joining rules in Table 2.2 can play their role in splitting (tracking) the permission flows through the system and in parallelizing the execution of sequential programs. This is also evident from the study of the existing permission-based program parallelization approaches (Aldrich et al., 2012; Stork et al., 2014), also note the permission flow graph (Figure 2.6) in Section 2.3.
Figure 4.14 shows the method call concurrency graph for the annotated program given in Listing 4.1, following the access permission splitting and joining rules. The analysis reveals that, in total, 11 out of 15 method calls across three classes can be executed in parallel, based on the access permission contracts. It also shows the effectiveness of generating permissions on the individual field of an object that can help achieve concurrency at a more granular level than generating permissions on the whole object. Recall the Pulse concurrency analysis for the same program in Section 4.3.1 that reports 4 out of 11 methods can be executed in parallel. This is because in Pulse, the permission contracts are generated at the object level. The constructor methods `ArrayCollection()` and `ObjectClass()` cannot be parallelized with any other method, as no other method can use an object before it is being instantiated with `unique` permission. However, constructors instantiating different objects can be executed at the same time. The graph shows that the methods that require either read permissions (`pure` or `immutable`) such as `printColl(...)` and `findMax(...)` on the same (different) objects can be executed in parallel. However, methods that require `full/share` (write) permission on a shared object e.g. `incrColl(...)` cannot be executed in parallel with other methods. This is also evident from the extended concurrency analysis presented in Section 4.3.1. Similarly, methods that require `unique` permission on the referenced objects such as `tidyupColls(...)` should always run in isolation.

Document Generation. Moreover, the `Sip4J` framework automatically generates a user-friendly report in the pdf format, comprising the correctness and concurrency analysis of the inferred specifications with the annotated version of the input Java program. The report can be used to provide developers a quick, abstract and explicit view of the implicit relations (data flow and alias flow) between objects at the method level and its concurrent behavior without looking at the source code. The generated report can be used by both novice and expert programmers (verifiers) with equal ease to analyze program behaviour.

The automatic inference of access permission contracts and its concurrency analysis by the `Sip4J` framework, along with its integration with the model checking tool, opened a new window for the program verifiers to automatically evaluate the desired behavior of large and complex applications, without any extra effort (annotation overhead) while consuming less time.

4.5 Threats to Validity

The `Sip4J` framework statically extracts the access permission contracts from the source code of a Java program. The objective is to make the implicit dependencies explicit thereby, the side effects in the program. The computed dependencies are then used to verify program behavior and automatically compute potential for concurrency in a sequential program.

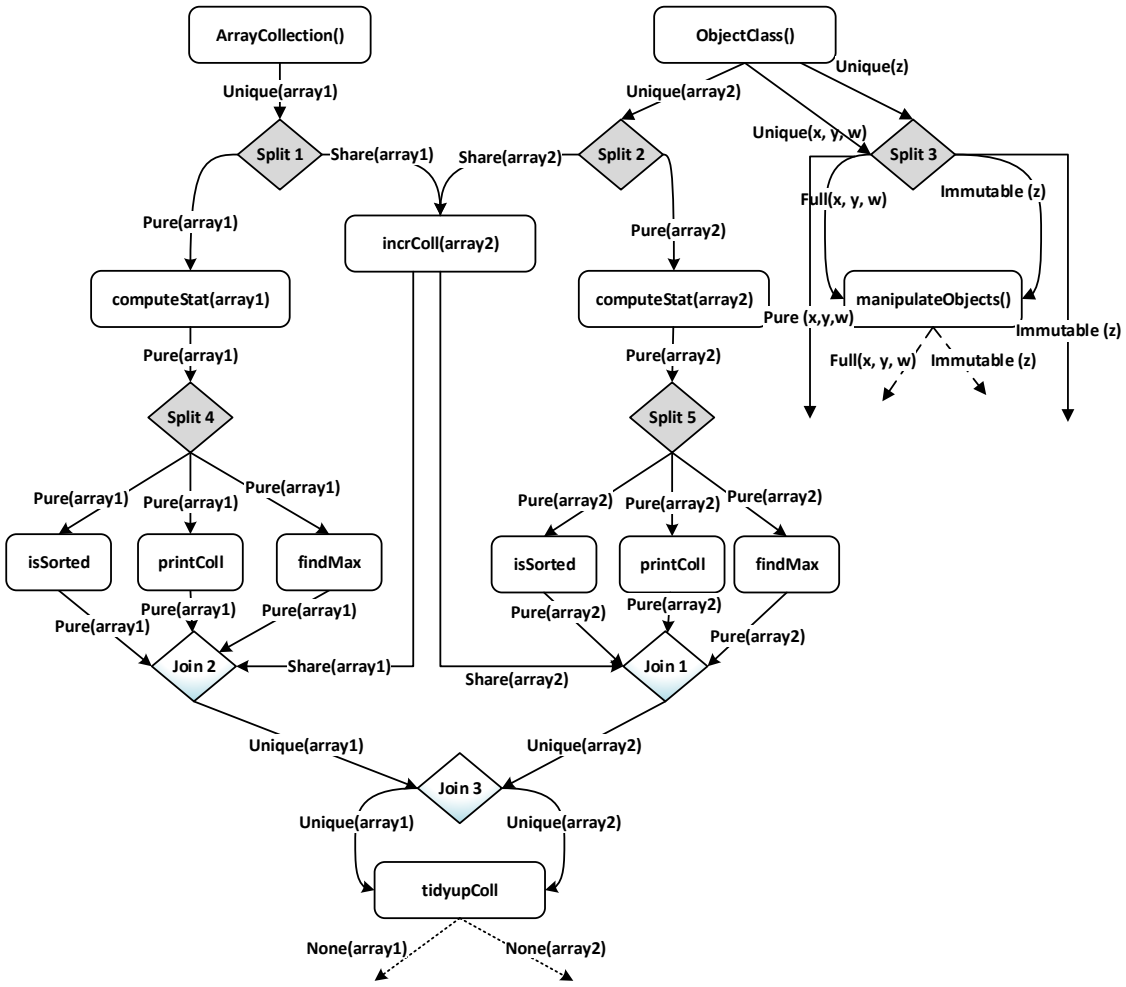

Figure 4.14: The method call concurrency graph of the example program given in Listing 4.1.

However, we identified some limitations of the Sip4J framework and threats to the validity of the underlying permission extraction and checking mechanism itself.

Construct Validity. The permission extractor in `Sip4J`, at the moment, does not support all the language constructs in Java Specification Language such as `method overriding (polymorphism), generics and lambda` expressions. This is because the technique is based on statically analyzing the source code. However, in the case of assignment statements, the analysis determines the actual instance type (the type of the object on the right-hand side) assigned to a reference of base type on the left-hand side of the assignment statement. This information can eventually be used to determine which overridden method should be parsed as a result of the method call, without executing the program. Moreover, our evaluation (Chapter 5) on small to large sized benchmark programs validate that through `Sip4J`, we have provided a succinct methodology to infer access permissions for realistic Java programs, implementing rich constructs of Java Specification Language.

Internal Validity. In certain situations, the permission inference framework does not pro-

duce the optimal (precise) solutions and generates safe (restrictive) permissions i.e., `unique` or `full` instead of `immutable` or `pure`.

This happens when a) an object is accessed in a complex infix expression with nested pre-fix or post-fix expressions or b) an object is accessed in a library method call expression. The first case is because the analysis, as explained previously in Section 4.2, is based on the type of expressions such as `Field_Access` and `Assignment` encountered in the `AST` of program, therefore, the permission extractor does not individually parse complex `infix-postfix` expressions as a part of the implementation. The second case is due to the unavailability of method definitions (bodies) for the library methods.

We believe this is an engineering problem in the `Sip4J` tool and generating restrictive permissions will not affect the integrity of the program itself when actually used for program verification or parallelization purpose. The restrictive permissions can be used to decide if the method can be executed in parallel with other methods in a safe way.

External Validity. It is worth noting here that the Sip4J framework, at the moment, does not automatically check the correctness of the inferred specifications for the overloaded constructors (methods) and consequently, it does not include them as a part of its concurrency analysis. We observe, the same is true for the parameter annotations, generated as a part of the method's signatures, in the Pulse translated version of the input program.

- This is because the Pulse tool does not support overloaded methods and parameter annotations as a part of its correctness and concurrency analysis. Consequently, the `Sip4J` framework excludes the overloaded constructors (methods) from the Pulse translated version of the program and analyzes them manually. However, the actual count of the concurrent methods may be misleading (underestimated) for programs having overloaded methods.

  In the case of parameters, the permission checking approach in the `Sip4J` framework, as mentioned previously in Section 4.2, solves the problem by mapping all the parameters with their actual referenced objects that are then automatically checked through the Pulse tool.

- It is worth mentioning here that concurrency analysis of the input specifications in the Pulse tool is performed at the class level, which makes it difficult for the `Sip4J` framework to check the potential for concurrency at the project level. However, it does not invalidate the effectiveness of our permission inference approach, in generating the correct specifications without annotation overhead as the problem is due to the inherent limitation of the Pulse tool itself.

  We believe all the problems discussed above are due to the inherent limitation of the Pulse tool and are engineering problems that can be solved, by extending the Pulse analysis with overloaded methods, and parameter annotations. However, in both cases, we manually check the inferred specifications as explained in Chapter 5.

Termination Analysis. The termination of the permission inference analysis in the `Sip4J` tool is based on the following characteristics.

- The analysis is based on the `AST` of the source code, a parse tree having a finite number of nodes (sub-expressions) for an expression statement. Therefore, it takes a finite number of steps to parse the generate tree with a finite number of expressions. Therefore, the proposed

analysis always converges to an expression type, designated as the base case, or otherwise terminates successfully.

- The presence of indirect or chained recursion, as explained previously in Section 3.2.1, can cause indefinite (infinite) loops and consequently, the memory overflows. This is because we save the meta-data (at-least its signature) of each method, and its current state, before parsing its body and switching the control to parse sub-method calls in it. This step helps to identify the second level (indirect) recursive method call for the same method during parsing, and ensures that analysis terminates successfully. Moreover, we validated through experiments, using realistic benchmarks, that the proposed analysis always terminates successfully.
- Another threat for the successful termination can be the <self-address-flow> statements (when a reference directly or indirectly starts pointing to itself creating a loop for analysis), the technique identifies such expressions following the pre-defined syntactic rules, (Appendix 1) for code parsing and terminates successfully, without creating any loop or cycles in the graph construction and traversal process eventually.

Soundness Analysis. In the context of permissions, the analysis is sound if it generates the required or sufficient permissions for each memory location that a method needs to read or write on during its execution.

- To validate that permission inference mechanism, the `Sip4J` tool always generates the required or sufficient, we call it safe permission, permissions. The analysis is supported by the pre-defined mathematically specified rules (Appendix 1), that try to capture all possible expression types in the Java Specification Language
- Further, the underlying approach ensures to minimize the number of false positives, which means that the `Sip4J` framework always generates permissions where required, by following the same set of syntactic rules, and never generates unwanted permissions.
- Moreover, evaluation (Chapter 5) of the `Sip4J` framework on realistic Java applications and its proof-of-concept by integrating the state-of-the-art model checking approach, Pulse, demonstrates the soundness (at least precision) of the underlying approach in generating the correct and required permissions.

4.6 Space and Computational Complexity Analysis

Space Complexity is the sum of the sizes of all the algorithm's data types as a function of input variables. It is expressed in terms of the function of input size and extra storage.

Let $\mathbf{S(n) = max(input, extra - storage)}$ denote the space complexity of permission inference algorithm.

For a Java program with C number of classes and M number of methods, the space complexity of M methods depends on the space complexity analysis of all the individual methods, as shown in Equation 4.1.

$$\textbf{Space(M)} = \sum_{m \in \mathbf{M}} Space(m) \tag{4.1}$$

The space complexity of each method m is the maximum space occupied by the local and global references in a method and the extra space it uses to store the information in a data structure, as shown in Equation 4.2.

$$\textbf{Space(m)} = \textbf{max}(\textbf{LocalSpace(m)} + \textbf{GlobalSpace(m)} + \textbf{ExtraStorage(m)}) \tag{4.2}$$

Therefore, the space complexity of a method m depends on the number of global reference variables ($RV(m)$), it accesses from its global environment, the number of local reference variables ($LRV(m)$) and the number of parameters ($P(m)$) in a method's signature. It further depends on the number of method calls (N) in each method. Further, it also considers the total number of aliases (RA) against each reference variable in a method.

The space complexity of each method can be calculated as follows:

- **LocalSpace(m)** = O(LRV(m) + P(m)) = the space occupied by the local variables and parameters.
- **GlobalSpace(m)** = for each i $\in RV(m)$, $Space(i)$ = $O(1)$ = the space occupied by the global references = a single word of memory allocated in the stack frame.
- **ExtraSpace(m)** = $2 * (O(\text{RV(m)} * \text{RA})) + 2$ = the total storage in data structure.
- If N is the total number of method calls inside a method, then
  - for each $i \in \mathbf{N}$, **Space(i)** = max(O($LRV(m) + P(m)$), 2 *(O($RV(m)$ * RA)))

Therefore,

$$\textbf{Space(m)} = \textbf{max}(\textbf{O}(\textbf{LRV} + \textbf{P(m)}) + \textbf{O}(\textbf{Space(N)}), (\textbf{2} * (\textbf{O}(\textbf{RV(m)} * \textbf{RA})) + \textbf{2}))$$

Hence, the space complexity of M methods in a program is calculated as follows:

$$\textbf{Space(M)} = \sum_{m \in \mathbf{M}} (max(O(LRV(m) + P(m)) + O(1) + O(Space(N)), (2 * (O(RV(m) * RA)))) \tag{4.3}$$

Similarly, the computation complexity analysis of an input program with M methods depends on the total number computation steps it takes to perform the analysis. Therefore, the computational complexity of our permission inference approach depends on the data-flow and alias-flow analysis ($DFAA(M)$) and context analysis $CA(M)$ of the source code, to fetch all the referenced objects (RV) accessed in (M) methods, along with their graph construction ($GC(M)$) and graph traversal process ($GT(M)$) as shown in Equation 4.4.

$$\textbf{CA-Perm(M)} = DFAA(M) + CA(M) + GC(M) + GT(M) =$$

$$\begin{aligned}
&\sum_{m \in \mathbf{M}} ((M + (P(m) * (N * A(n))) + RV(m)) \\
&+ (ES * (LRV(m) + (RV(m) * (P(m) + RA)))) \\
&+ DFAA(N) \\
&+ (RV(m)^2) \\
&+ (RV(m)))
\end{aligned} \tag{4.4}$$

The detailed computational complexity analysis of the permission inference approach presented in this thesis is given in Appendix 2.

Chapter 5

Evaluation

This chapter presents the evaluation of the Sip4J framework, along with its integration with the Pulse tool to verify the correctness of the inferred permissions. The empirical evaluation was performed on realistic Java applications using four benchmarks suites such as Java Grande benchmark (jomp)[1], Æminium[2] and Plaid[3] and Crystal[4], together with Pulse[5] itself. The evaluation confirms that the `Sip4J` framework is indeed capable of generating the permission-based specifications from the source code of realistic Java programs in a correct and efficient way. It further presents the efficacy and effectiveness of the proposed framework by summarizing the results of the evaluations.

5.1 Evaluation Criteria

The evaluation of the permission inference framework and the inferred specifications is based on the following criteria and experiments, where we:

1. validate the **correctness** of the inferred permissions in two ways (Section 5.4) by
   - automatically checking the inferred specifications using the Pulse tool (Section 5.4.1) and,
   - manually generating the specifications by looking at the source code and comparing them with the one inferred by the Sip4J as well as with the results produced by the Pulse tool (Section 5.4.2).
2. demonstrate the **effectiveness** of the inferred permissions (Section 5.5) by performing the **concurrency analysis** of the inferred specifications by the Sip4J framework as an extension to the Pulse tool.
3. demonstrate the **effectiveness** of permission inference technique itself (Section 5.6) by computing the **number of annotations** generated by the Sip4J to measure the annotation overhead (effort saved) and its scalability for realistic programs.
4. perform the **efficiency analysis** of the permission inference technique by computing the

[1] https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite
[2] https://github.com/AEminium/AeminiumBenchmarks/tree/master/src/aeminium/runtime/benchmarks/.
[3] https://github.com/plaidgroup/plaid-lang.
[4] https://code.google.com/archive/p/crystalsaf/
[5] http://aeminium.dei.uc.pt/index.php/ToolsAndDownloads

execution time of the permission inference analysis to automatically generate specifications (Section 5.7).

5.2 Experimental Setup

All experiments were performed on MacBook Pro, Intel Core i7, (2.3GHz) processor (4 physical cores) and 16GB of RAM. The development environment includes Eclipse IDE 3.7.2, JDK 1.7 and Antlr compiler 3.3 and TexLive 2015.

5.3 Datasets

The dataset for the evaluation consists of benchmark programs and realistic Java applications widely used in the research community (Bull et al., 2000; Aldrich et al., 2011; Cataño et al., 2014; Aldrich et al., 2012; Stork et al., 2014; Fonseca et al., 2016) to evaluate the permission-based program verification approaches and to gain performance improvements in automatic parallelization approaches. A brief characteristic of the benchmark programs is given below.

1. **Java Grande Benchmark.** It is a Java Grande benchmark suit (Bull et al., 2000). The applications for evaluation are mainly picked from the simple Kernels (Section II) and Applications (Section III) section of the benchmark. The chosen programs are large scale data and computation intensive programs such as `montecarlo`, `moldyn`, etc. aimed at testing the performance improvement of sequential programs through the Java execution environment. For example,

MonteCarlo is a financial simulation, using the Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data.

Moldyn is an N-body code modeling particles interacting under a Lennard-Jones potential, in a cubic spatial volume with periodic boundary conditions. Performance is reported in interactions per second. The number of particles is given by N. The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair-wise manner.

Euler solves the time-dependent `Euler` equations for flow in a channel with a `'bump'` on one of the walls. The structure employs an $N*4N$ mesh, and the solution method is a finite volume scheme using a fourth order `Runge-Kutta` method (Abdel Karim, 1966) with both second and fourth order damping. The solution is iterated for 200 time steps. Performance is reported in units of time steps per second.

Search solves a game of $connect-4$ on a $6*7$ board using an alpha-beta pruned search technique. The problem size is determined by the initial position from which the game is analysed. The number of positions evaluated, N, is recorded, and the performance reported in units of positions per second.

Series computes the first N `Fourier coefficients` of the function $f(x)=(x+1)^x$ on the

interval 0,2. This benchmark heavily exercises transcendental and trigonometric functions.

LuFact Solves an `N * N` linear system using *LU* factorization followed by a triangular solve. This is a Java version of the well known Linpack benchmark (Dongarra, 1992).

SOR performs 100 iterations of successive over-relaxation on an `N * N` matrix. The performance reported is in iterations per second.

Crypt performs IDEA (International Data Encryption Algorithm (Lai et al., 1991)) encryption and decryption on an array of N bytes. Performance units are bytes per second.

Matmultsparse uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirect addressing and non-regular memory references. An `N * N` sparse matrix is multiplied by a dense vector 200 times.

2. **Æminium Benchmark.** All the programs in the Æminium benchmark consist of data and computationally intensive applications that mostly solve problems using `Arrays` and `Collection` data structures in Java such as `de Columbian Health Care System` and `gaknapsack`. The applications were taken from different sources such as HPCC (High Performance Cluster Computing)[6], BOT (Barcelona OpenMP Tasks Suite)(Duran et al., 2009) and Java ForkJoin framework (Lea, 2000). All the applications in this benchmark are part of Æminium[7] project, a by-default concurrent programming paradigm, that has been used to evaluate the performance of a single-threaded program on multi-core processors, by parallelizing its execution based on access permissions. For example,
Gaknapsack solves the Knapsack problem with 500 items using a Genetic Algorithm, with the population size of 10,000 generations, a probability of mutation and recombination of 20.

Health simulates de Columbian Health Care System (Das and Fujimoto, 1993). It uses multilevel lists where each element in the structure represents a village with a list of potential patients and one hospital. The hospital has several double-linked lists representing the possible status of a patient inside it (waiting, in assessment, in treatment or waiting for reallocation). At each time step, all patients are simulated according to the several probabilities (of getting sick, needing a convalescence treatment, or being reallocated to an upper-level hospital). A task is created for each village being simulated. Once the lower levels have been simulated synchronization occurs.

BlackScholes is a MonteCarlo simulation using the BlackScholes Formula with 10,000 points.

QuickSort sorts an array of 100000 random long numbers using quicksort algorithm.

3. **Plaid Benchmark.** The data-set common in the Æminium and Plaid benchmark mostly consists of computationally intensive applications implementing recursive data structures and divide-and-conquer algorithms such as.
FFT is an application (used for weather forecasting) that performs a `Fast-Fourier Transform` on a large one-dimensional array of 1048576 random complex numbers using the

[6] https://icl.utk.edu/hpcc/
[7] http://aeminium.dei.uc.pt/index.php/AEminium

generalized Cooley-Tukey algorithm[8].

Integral is a computation-intensive application (library) developed by Æminium. It computes the integral of a user-defined function. The integral is computed by dividing the overall interval into infinitesimal small intervals, calculating the approximate area for all the subintervals, and then all fractions added to compute the area of the whole integral.

Webserver is a client-server application for serving static web pages. In the Æminium project, it was used to parallelize code by overlapping the computation and communication over multiple pages to improve performance.

Fibonacci computes fibonacci of number 25.

ShellSort sorts an array of 100000 random long numbers using shellsort algorithm.

4. **Pulse** is a permission-based model checking tool, implemented as a Java Eclipse Plugin. The tool was developed, as a part of the Æminium project [9], to automatically perform access permission checking and verify program behavior based on the specifications. The motivation behind using the Pulse tool itself as a case study was to evaluate the permission inference technique on a real-time Java application of average size, in this case, its 7k plus SLOC. Pulse is an object-oriented program that implements rich Java constructs such as classes, inheritance, method overloading, regular expressions, and it extensively uses Java APIs, especially the multi-level linked lists to perform static analysis of the source code with the input specifications, to generate its abstract state-machine model that is then verified by a symbolic model checker.

5. **Crystal** is a static analysis framework built as an Eclipse Plugin in Java, developed at Carnegie Mellon University for teaching and research purposes. The static analyzer Crystal is an object-oriented Java application, developed as a part of the Plural project [10]. Plural is a sound modular protocol checking tool that employs access permissions to allow a flexible aliasing control mechanism to verify program behavior. Crystal performs branch-sensitive data-flow and exceptional control-flow analysis of the Java source code. The data-flow analysis is based on a "worklist" algorithm. The analysis extracts different information at different program levels to perform program verification based on the input specifications. The motivation behind using the Crystal applications is to evaluate the permission inference mechanism on a realistic big case study, in this case, 17k plus lines of the source code, that follows object-oriented concepts such as object encapsulation, object composition, object aggregation, etc. Moreover, it implements rich Java Specification Language constructs such as classes, anonymous classes, inner classes, enumeration classes, abstract classes, method overloading, method overriding, multiple inheritance through interfaces, generics and exceptional handling.

A brief statistics of the data sets are given in Table 5.1.

The results of the experiments on the benchmark programs are discussed below.

[8] https://sourceforge.net/projects/hpcc/
[9] http://aeminium.dei.uc.pt/index.php/AEminium
[10] https://code.google.com/archive/p/pluralism/

Table 5.1: A brief statistical view of the benchmark programs.

		Statistics		
Benchmark	**Program**	**SLOC**††	**Classes**	**Methods**
Plural	Crystal	17,512	212⋆	1,975
Pulse	Pulse	7,671	40◇	461
jomp	montecarlo	1,370	18	196
	euler	1,080	7	51
	search	666	7	50
	moldyn	608	7	43
	lufact	549	5	42
	crypt	488	5	40
	series	359	5	37
	sor	354	5	34
	sparsematmult	327	4	33
Æminium	gaknapsack	437	6	21
	blacksholes	232	6	50
	health	232	6	18
Plaid†	webserver	143	3	12
	fft	91	4	11
	quicksort	66	3	9
	shellsort	58	2	7
	integral	40	1	5
	fibonacci	22	1	4
Example	ArrayCollection	71	3	12
Total		32,376	**350**	**3,111**

†† The SLOC are computed using the SLOCCount tool, https://dwheeler.com/sloccount/

⋆ It includes 22 anonymous classes plus 66 interfaces.

◇ It includes three inner classes.

† The .java version of the program common in the Æminium and Plaid benchmark.

5.4 Correctness analysis of the Inferred Specifications

The correctness analysis of the inferred specifications is performed in two ways a) automatic analysis (Section 5.4.1) through the Pulse tool and b) manually generating the specifications for all the benchmark programs.

5.4.1 Automatic Analysis

The automatic correctness analysis of the inferred specifications is performed by integrating the Pulse tool as a part of Sip4J framework as explained in Section 4.3. Pulse verifies the correctness of the input specifications by performing the method (un) satisfiability analysis following the CTL formula given in Section 2.2.2.

Evaluation Metrics

The automatic correctness analysis of the inferred specifications is based on the following metrics in Pulse.

- **#Satisfiable(M)** shows the number of methods determined by the Pulse tool to be satisfiable where a method is satisfiable if all its pre-conditions are met. The method satisfiability analysis is performed using the `requires` clause in the input specifications. In other words, method satisfiability analysis shows the reachable code (method). A method is reachable if it obtained its required permissions.
- **#UnSatisfiable(M)** shows the number of methods that Pulse determines to be unsatisfied or unreachable. The presence of the unsatisfiable method is due to the method's unsatisfiable pre-conditions which either indicates an error in the generated specifications or some missing specifications.

In other words, total methods are a sum of satisfiable and unsatisfiable methods for each program.

Analysis

Table 5.2 shows the correctness analysis of the benchmark for the Pulse translated (Plural annotated) version of each program. For every program (column **Program**) in the benchmark suit, we present the results for the correctness analysis (columns **#Satisfiable(M)**, **#UnSatisfiable(M)**) of each program with M methods.

The results confirm that our `Sip4J` framework successfully infers satisfiable (required or safe) permissions, without any specification errors, for all methods in all the benchmark programs with three exceptions: montecarlo, Pulse and Crystal. For the monetcarlo program, 11 of the 196 methods have been determined to be un-satisfiable. Upon manual analysis of the montecarlo source code. We noticed that unsatisfiability of the methods is due to the fact, as discussed previously in Section 4.5, that the Pulse tool does not support overloaded methods (constructors) as a part of its permission checking mechanism, and that all these 11 methods are overloaded methods. For Crystal, the analysis shows unsatisfiability for 581 out of 1,975 methods. Again, this is due to the presence of overloaded methods in the Crystal framework. However, we manually analyzed the overloaded methods with their inferred specifications and found them to be satisfiable.

5.4.2 Manual analysis

To check if the inferred specifications comply with the code, we a) generate access permissions by looking at the source code for each program at the method level and, b) compare the generated results with the specifications inferred by the Sip4J framework as well as with the result produced by the Pulse tool automatically.

By manually generating the specifications from the source code, we were also able to compute the number of safe approximations (e.g., `full` or `unique` permissions instead of `pure` or `immutable` permission) made by the Sip4J tool, as required in certain cases (the

Table 5.2: Correctness analysis of the inferred specifications.

	Correctness analysis		
Program	**#Satisfiable(M)**	**#UnSatisfiable(M)**	**#SafeApprox(M)**
Crystal	1394	581[7]	288[4]
Pulse	451	10[7]	13
montecarlo	185	11[7]	10
euler	51	0	15
search	50	0	10
moldyn	43	0	10
lufact	42	0	10
crypt	40	0	10
series	37	0	10
sor	34	0	11
sparsematmult	33	0	11
blacksholes	50	0	0
gaknapsack	21	0	0
health	18	0	0
webserver	12	0	0
fft	11	0	0
quicksort	9	0	0
shellsort	7	0	0
integral	5	0	0
fibonacci	4	0	0
ArrayCollection	12	0	0
Total	**2,509**	**602**	**398**

[7] It shows the number of overloaded methods (constructors) in the program.

[4] It includes 83 safe approximations for generics (parameterized) methods in Java.

reasons are already explained in Section 4.5). However, while doing so, we observed that identifying dependencies in a small program was quite easy for a human. However, as the size and complexity of program grows, it became increasingly challenging to precisely identify and track the dependency information in the program and that supports the main motivation of the research presented in this thesis.

Evaluation Metrics

SafeApprox(M) shows the number of safe approximations generated by the `Sip4J`, for the Plural annotated version of each program for `M` methods. A non-zero number indicates the situation where our `Sip4J` does not produce an optimal solution but generates restrictive (`full` or `unique`) permissions as a safe option to avoid data integrity problems. The safe permission guarantees that the method got sufficient permissions to start its execution and this information can be used to identify the side effects methods produce on each other when executed in parallel.

Analysis

In Table 5.2, column # **SafeApprox(M)** shows the number of safe approximations (safe permissions) made by the `Sip4J` framework for the Pulse translated version of each program with `M` methods.

For example, in the case of Java Grande benchmark, all the programs use `JGFInstrument` and `JGFTimer` class libraries from the Java Grande Framework, with ten (10) library methods calls in each program for which `Sip4J` generates safe permissions. In the case of Crystal, the count is 288 as the benchmark is heavily dependant on the Java class libraries. It includes 83 generics (parameterized) methods.

In total, `Sip4J` made 398 safe approximations for a total of 3,111 methods for the Pulse translated version of 21 benchmark programs that is 4% of the total annotations (10,157) generated by `Sip4J`. However, we observed that generating safe permissions does not affect the integrity of the specifications and the program itself when actually used for verification or parallelization purpose, and it does not invalidate the effectiveness of our technique in automatically generating correct specifications.

Overall, the analysis shows the effectiveness of our proposed framework in automatically identifying and tracking such subtle dependencies correctly.

5.5 Concurrency Analysis of the Inferred Specifications

The Sip4J framework computes the potential for concurrency in a sequential program by integrating the Pulse tool and extending its concurrency analysis, based on five types of inferred permissions, as explained in Section 3.3.2 and then implemented in Section 4.3. It computes the number of method pairs that can potentially be parallelized with other methods or the methods that should always run sequentially based on the generated specifications. It is worth mentioning here that the concurrency analysis of the inferred specifications in the Pulse tool is based on satisfiable methods.

Evaluation Metrics

For every program in the benchmark suite, it computes the percentage of the method (pairs) that can run in parallel based on the following metrics,

- **Concur(M)(%)** shows the overall percentage of methods that could be parallelized with at least one other method (including method with itself) in the program with `M` methods whereas,
- **Concur(MP)(%)** is the percentage of the total number of method pairs that can be parallelized (including method with itself) over all possible pairs of methods for `M` methods.

Analysis

Table 5.3 shows the concurrency analysis of all the benchmark programs. For example, for the montecarlo program, the **Concur(M)(%)** analysis shows that 92 (49%) out of total 185

satisfiable methods could potentially be executed in parallel with at least one other method in the program.

The **Concur(MP)** ratio for the monetcarlo program is 20% for 185 methods. In other words, 3404 (20%) pairs of methods, with a state space of $\binom{185}{2}+185=17{,}020$ total pairs, can be parallelized in 18 classes. Using Pulse source code itself as a case study, the **Concur(M)(%)** ratio is 67% with 299 methods that can be run in parallel with other methods and the **Concur(MP)** ratio, in this case, is 34% for 451 methods.

We exclude overloaded methods from the Plural annotated version of the input program to perform its concurrency analysis, this is because the model-checking approach in the Pulse framework reports overloaded methods to be unsatisfiable as the correctness analysis in Pulse does not support overloaded methods (explained in Section 4.5). However, this limitation of the Pulse tool does not invalidate the effectiveness of `Sip4J` framework in generating the correct specifications. In this case, the results (count) of the concurrency analysis may be misleading (under-estimated) for the programs having overloaded methods. We observed that the exclusion of overloaded constructors, (e.g., 10 in the case of Pulse and 11 for the monetcarlo program), does not affect the potential for the concurrency as constructors cannot be parallelized with any other method during program execution.

In summary, the permission-based concurrency analysis in terms of the number of concurrent methods **(Concur(M)%)** vary from 35 to 62% for the Java Grande benchmark programs, 5 to 66% for the Æminium and Plaid programs and the ratio is 67% for the Crystal and Pulse itself when used as a case study.

5.6 Annotation Overhead

The permission inference approach avoids the annotation overhead in two ways by generating a) five kind of access permissions at the object's (class) field level and, b) permission contracts (Plural specifications) to perform evaluation by the Pulse tool. We compute the annotation overhead as a way to quantify the manual (annotation) effort by measuring (1) the amount of annotations generated by `Sip4J` on individual fields of an object, (2) the number of annotated lines of code (permission contracts) generated by the `Sip4J` framework for the Pulse translated version of the programs and, (3) the number of individual annotations for the Pulse translated version.

Evaluation Metrics

To compute the annotation overhead, we use the number of methods **M**, as the basis for evaluation as access permissions (contracts) are generated at the method level.

1. $\#\mathbf{AnnLOC(M)}_P$ shows the **annotated lines of code** generated for the Pulse translated version for **M** methods in a program. It counts one line (a permission contract N) for each method with a '`requires`' and an '`ensures`' clause, one line for each class, to define the typestate information at the class level, and one line to import the package to support the Plural annotations in a Java program. Therefore,

Table 5.3: Concurrency analysis of the inferred specifications

Program	**Concur(M)(%)**[8]	**Concur(MP)(%)**[8]
Crystal	944 (67%)	52%
Pulse	299 (67%)	34%
montecarlo	92 (49%)	20%
euler	18 (35%)	8%
search	18 (36%)	9%
moldyn	16 (37%)	10%
lufact	25 (59%)	17%
crypt	25(62%)	17%
series	21 (56%)	15%
sor	16 (47%)	11%
sparsematmult	16 (48%)	10%
blacksholes	9 (18%)	6%
gaknapsack	4 (19%)	6%
health	1 (5%)	1%
webserver	8 (66%)	55%
fft	5 (45%)	36%
quicksort	2 (22%)	11%
shellsort	2 (28%)	17%
integral	3 (60%)	20%
fibonacci	2 (50%)	30%
ArrayCollection	5 (41%)	33%
Total	**1,531 (61%)**	-

[8] It excludes concurrency analysis of the overloaded and parameterized methods.

$$\#AnnLOC(M)_P = \mathbf{N} + \mathbf{C} + 1 \tag{5.1}$$

where N shows the number of annotated lines (permission contract) generated for M methods in a program. The number of annotated lines (`N`) would be equal to the total methods `M` if permission contracts are generated for all the methods in a program.

2. # **Anns(M)**$_P$ calculates the number of **individual annotations** generated as a part of the permission contracts for `M` methods in the Pulse translated version of the program. The number of annotations, in this case, depends on the presence (absence) of annotations on the receiver object (`this`) and the number of parameters (aliases of the referenced objects) accessed in a method.

   For each *non-constructor* method accessing a field of the receiver object `(this)` it includes two annotations to specify pre- and post-permission on the object.

   For each *non-constructor method*, **Anns(M)**$_P$ counts two (as part of `requires` and `ensures` clause) to add a typestate '`alive`' as a part of pre- and post-permission for each class field and parameter (if any) at the method level.

   For each *non-constructor* method with parameters, **Anns(M)**$_P$ adds two (pre- and post-permission) for each parameter.

For each *non-constructor* method that returns a global object or alias of a global reference, $\mathbf{Anns(M)}_P$ adds two annotations as a part of `ensures` clause to specify the permission generated by a method on the returned object (R). Finally, $\mathbf{Anns(M)}_P$ counts one for the annotation `ENDOFCLASS` to mark the end of each class in a program (as required by Pulse).
For each *constructor method*, there is only one permission annotation for receiver object (`this`) and one typestate annotation. It can have only '`ensures`' clause as part of the permission contract in Pulse. There are no annotations for the parameters as Pulse does not support overloaded constructors with parameters.
Let M be the set of methods, $\mathbf{M}_C$ be the number of constructors, $\mathbf{M}^F_{NC}$ be the number of non-constructor methods that access some global (class) field directly or indirectly in a method, $\mathbf{M}^R_{NC}$ be the number of non-constructor methods that returns a reference object (i.e., $\mathbf{M}=\mathbf{M}_C+\mathbf{M}^F_{NC}) + \mathbf{M}^R_{NC}$. $\mathbf{Anns}_P$ is defined as:

$$Anns(M)_P = (1+1) * \mathbf{M}_C + (2+2)*\mathbf{M}^F_{NC} + \sum_{m \in M} (2 * P(m)) + 2 * \mathbf{M}^R_{NC} + \mathbf{C} \tag{5.2}$$

where $P(m)$ denotes the number of parameters (aliases of some global object) accessed in method m.

3. $\#\mathbf{Anns(M)}_F$ calculates the number of **individual annotations** generated as pre- and post-permissions at the object's (class) fields for `M` methods in a program. This includes two annotations (generated as post-permission) on the returned object (R) by a non-constructor method. Therefore, the number of permission-based annotations generated at the field level for **M** methods is

$$Anns(M)_F = \sum_{m \in M} (2 * F(m)) + 2 * \mathbf{M}^R_{NC} \tag{5.3}$$

where $F(m)$ is the number of individual fields and its aliases accessed in method m and $\mathbf{M}^R_{NC}$ be the number of methods that returns a referenced object.

Analysis

Table 5.4 shows the results (columns $\#\mathbf{AnnLOC(M)}_P$, $\#\mathbf{Anns(M)}_P$) of the effectiveness analysis of our technique for the benchmark programs **Program**).

For example, for the `ArrayCollection` example program given in Listing 4.1 with 3 classes and 12 methods, `Sip4J` generated 15 annotated lines of code (`AnnLOC`) with a total of 78 individual annotations for the Pulse translated version of the program, where permissions are generated on the receiver object and 49 annotations when permissions were generated at the field level. Likewise, for Crystal, `Sip4J` generated 2,188 annotated lines with a minimum of 5,234 annotations, for 1,975 methods for the Pulse translated version of the program and the count was 6,691 when permissions are generated at the field level.

We observe that the use of parameter annotations causes the state space explosion problem for big programs such as Crystal. This is due to model checking the specifications

Table 5.4: Effectiveness and efficiency analysis of the permission inference technique.

	Annotation Overhead			Performance
Program	# **AnnLOC**$_P$[9]	# **Anns(M)**$_P$[9]	# **Anns(M)**$_F$	**Time(M)** (sec)
Crystal	2,188	5,234[10]	6,691[10]	1441.056
Pulse	513	1,764	4,850	41.84
montecarlo	204	948	1,360	17.95
euler	52	197	1,073	5.42
search	60	204	691	1.41
moldyn	52	205	901	1.40
lufact	50	325	437	1.13
crypt	46	163	385	0.98
series	43	143	207	0.56
sor	42	153	267	0.61
sparsematmult	36	156	316	0.51
blacksholes	56	250	694	1.92
gaknapsack	38	82	250	1.03
health	25	74	334	0.30
webserver	11	25	11	0.24
fft	16	56	44	0.43
quicksort	13	33	17	0.06
shellsort	10	32	44	0.06
integral	7	17	17	0.08
fibonacci	8	18	9	0.03
ArrayCollection	15	78	49	0.16
Total	**3,485**	**10,157**	**18,647**	**1,517.47**

[9] The parameter notations were excluded for the Pulse analysis.

[10] It includes 2,324 annotations for the 581 overloaded methods with at-least 4 annotations per method.

in the Pulse tool that creates a big state space due to a large number of annotations. As discussed previously in Section 4.5, we omit the parameter annotations, from the Pulse translated version of the programs, to automatically verify the correctness of the inferred specifications through the Pulse tool that also avoids the state space explosion problem due to the presence of parameter annotations in the input specifications.

However, we manually counted the number of annotations for the parameters to quantify the programmers' effort, to manually add permission-based specifications in the program. The amount of annotations generated shows the minimum annotation overhead, with a single typestate, posed by the existing permission-based verification approaches such as Plural and Pulse itself.

The analysis shows the effectiveness of the permission inference framework in automatically generating the specifications which otherwise need to be manually identified and added in the program.

5.7 Performance Evaluation

We compute efficiency of the permission inference algorithm by measuring the average execution time of the analysis in seconds, averaged over 10 independent runs on Java Virtual Machine.

Table 5.4, column `Time(M)`, shows the execution time (in seconds) of the analysis to automatically generate the permission-based dependencies from the source code for `M` methods in a program. The result shows that it takes about 18 seconds to generate specifications for the montecarlo program with 196 methods, and a fraction of a second for programs in the Æminium and Plaid benchmarks being smallest in size. For Crystal, the biggest case study we evaluated with 17K plus physical source lines of code, it took 1441.056 seconds (about 25 minutes) to generate 2,188 annotated lines of code, with a minimum of 5,234 annotations, for 1,975 methods for the Pulse translated version of the program.

The results of analysis shows that the permission inference framework not only avoids the developer's (manual) effort in terms of annotation overhead, but saves their time to explicitly specify permission-based specifications in the source program.

5.8 Evaluation Summary

All the experiments confirm that our inference approach is indeed capable of generating the permission-based specifications from the source code of realistic Java programs in a correct and efficient way.

The results of the correctness analysis show the potential of the Sip4J framework to generate the required or safe permissions for all the benchmark programs. However, in some cases, the framework does not produce the optimal (required) results but created the safe permissions as explained in Section 4.5. One such situation was the handling of the library methods where we don't have the definition of the method for which analysis generated the safe (write instead of read) permissions on the referenced objects to ensure the integrity of the data.

Similarly, the concurrency analysis by the `Sip4J` framework confirms the potential in the inferred specifications to enable concurrency for sequential programs, which was one of the motivations of our work.

The annotation overhead analysis shows the effectiveness of the inference technique in automatically generating the specifications which otherwise need to be manually identified and added in the program and that pose a significant annotation overhead for the programmers. In total, `Sip4J` generated 18, 647 permission annotations at the field level, for 3, 111 methods in 350 classes for 21 programs and 3,485 annotated lines were generated for the Pulse translated versions of the same programs, with 10, 157 minimum annotations in less than 1,517 seconds (about 26 minutes).

If created manually, the above annotations may take time that could be multiples of months, as in the case of Pulse Cataño et al. (2014) itself, where authors reported that they took six months to manually identify dependencies and annotate a Java application, (14k

SLOC, 55 classes and 376 methods), with 546 lines of permission-based specifications to verify its behavior. No doubt, the complexity and design of the input program does have an impact on the annotations generation time. However, we expect that the execution time of the inference process in the `Sip4J` can be a multiple of minutes, as in the case of Crystal application (17k plus SLOC, 212 classes, 1,975 methods), or can be in hours in the worse-case for really large applications, but not in months, thus showing the effectiveness of our technique in generating permission-based annotations while saving time.

Chapter 6

Related Work

This chapter provides a review of the state of the art permission-based program verification and parallelization approaches. The study explains the pragmatics of access permissions starting from the Boyland's fractional permission sharing model to all of its variants (counting and symbolic permissions) while comparing and contrasting them in different approaches. However, in this research, our aim is to review the symbolic permissions with a special focus on their usage in the existing approaches to verify the program behavior and parallelize their execution.

Correctness and reliability of software programs written in imperative and object-oriented languages such as Java and C++ have always been a major challenge for the IT industry. This is because of the implicit dependencies that exist between the code and program states. In imperative programs different program parts may access the same mutable state, without exposing this information to each other and, consequently, may cause unwanted interference or inconsistent states. Preventing such errors is important to ensure the compliance of API (Application Programming Interface) protocols in object-oriented programs and to verify the correctness of increasingly ubiquitous multi-threaded applications.

Modern object-oriented programs are highly reliant on reusable APIs that often define usage protocols i.e., the desired sequence of method calls that API clients must follow for underlying objects to work properly. Typestates (Strom and Yemini, 1986) have been designed to specify usage protocols and verify their behavior. A typestate abstractly defines an object's state at execution time. However, statically tracking object state is a non-trivial task because of unexpected transitions between states during program execution. In multi-threaded programs, managing synchronization between threads is a complicated and challenging task for programmers due to thread interleaving and heap interference, which can lead to common concureency problems such as deadlocks, data races. The situation becomes worse in the presence of unrestricted aliasing, the hallmark feature of imperative and object-oriented languages (Bierhoff et al., 2009b).

Earlier work on verifying correctness of sequential programs dates back to Hoare's logic (Hoare, 1969) that defines a set of axioms (calculus) and formal inference rules to specify and verify desired properties and formulate static analysis technique (Floyd, 1993). Handling thread non-interference for Java-like concurrent programs dates back to Owicki-Gries' axiomatic method, (Owicki and Gries, 1976), and Jones' rely-guarantee principle (Jones, 1983). These approaches are considered the traditional and general ways of performing shared-memory program verification. Since the seminal work of Hoare (Hoare, 1969) and

Floyd (Floyd, 1993), many logic-based verification approaches and type-effect systems have been developed to avoid problems such as data races, deadlocks, atomicity violations in shared-memory programs.

The commonly used verification approaches either perform deductive verification or employ theorem proving techniques where formal correctness proofs are used to verify program properties based on the input specifications ((Huisman, 2001; Flanagan et al., 2003; Abadi et al., 2006; OHearn, 2007; Villard et al., 2010; Caires and Seco, 2013)). Others conduct dynamic analysis of the input program through model checking (Visser et al., 2003; Chaki et al., 2004; Chaki and Gurfinkel, 2018) or perform static analysis (Boyapati et al., 2002; Engler and Ashcraft, 2003; Voung et al., 2007; Naik et al., 2009; Dias et al., 2013) to approximate runtime behavior of the program to verify its correctness.

However, in the last decades, static contract checking based on Hoare logic and the development of advanced, simplified, automated program verifiers (Fähndrich and Logozzo, 2011; Pradel et al., 2012; Filliâtre and Paskevich, 2013; Carr et al., 2017) had been an active areas of research for the verification of program behavior. Although these approaches are usable and quite promising, the support for concurrency and aliasing is limited. As most of the real-world applications are inherently multi-threaded, the next step was to develop tools and techniques that can reason about shared-memory programs and control aliasing in a sound and efficient way. To this end, permission-based program logics and tools became influential. Access permission, a novel abstraction, provides strong reasoning mechanisms to handle both aliasing and concurrency.

Access permission, formally called Boyland's fractional permission (Boyland, 2003), is a formalism inspired by Linear Logic (Girard, 1987) and Separation Logic (O'Hearn et al., 2001; Reynolds, 2002). The former treats permissions as linear resources and the latter simplifies the specifications and verification of shared-memory programs in an efficient way. Fractional permissions were originally proposed to verify non-interference of the program states in parallel programs, using either read or write accesses on the referenced objects. The formalism was later extended by Bornat et al. (2005) to allow read sharing of the shared program states. Bierhoff and Aldrich (2007) extended fractional permissions as symbolic permissions to model both the read/write operations and aliasing information of a program state at one place.

A study of the literature shows that access permissions have been investigated to address different concerns related to security, concurrency, and protocol verification. Notable threads of research include Plural (Bierhoff and Aldrich, 2007, 2008; Beckman, 2009), Chalice (Leino et al., 2009; Leino and Müller, 2009), VeriFast (Jacobs et al., 2011), VPerm (Le et al., 2012), Pulse (Siminiceanu et al., 2012), Sample (Ferrara and Müller, 2012), Plaid (Aldrich et al., 2012), Æminium (Stork et al., 2014), VerCors (Amighi et al., 2012, 2014), and Viper (Müller et al., 2017), to name a few.

The state of the art is categorized along the following four dimensions:

1. Verification of API protocols in typestate-based sequential and concurrent programs.
2. Verification of common concurrency problems such as race conditions and deadlock etc.,

in concurrent programs.

3. Automatic inference of access permissions from sequential and concurrent programs.
4. Access permission-based automatic parallelization of sequential programs.

Within each of the above categories, the existing approaches are compared and contrasted based on the following criteria:

a) The type of underlying program (`Prog`) such as sequential or concurrent program.

b) The programming language (`Language`) used or developed as a specification language.

c) The realization of the proposed technique in the form of a tool (`Tool`).

d) The type of analysis (`Analysis`) i.e. static or dynamic performed.

e) The kind of permission abstraction (`Perm-Kind`) such as fractional, counting or symbolic permissions, supported as a part of specifications.

f) The access notations or contracts (`Perm-Specs`) specified as program annotations.

g) The permissions or access notations (`Perm-Infer`) inferred.

h) The annotation overhead (`Anno`) (if any) posed by the approach.

i) The functional or behavioral properties (`Properties`) verified, based on the permission-based specifications.

The rest of this chapter is organized as follows. Section 6.1 briefly discusses the related formal theories and type systems in the literature as seminal and background work to permission-based program verification and parallelization work. Sections 6.2, 6.3, 6.4, 6.5 covers the four categories of the state of the art mentioned previously. Some of the approaches are elaborated with sample programs to show how different types of permission annotations are used at the code level. Further, in every section, a brief summary of the studied work in chronological order (where possible), following the above mentioned criteria, is given in the form. Finally, section 6.6 provides an insight into the use of access permission-based specifications and the research challenges posed by the existing approaches.

6.1 Related Formalisms to Program Verification and Parallelization

This section briefly presents other formal type theories for program verification and parallelization. A type system can verify the desired interaction between system components as types can classify program entities and the permissible results of the computations. The beauty of type systems is the assurance that if a program is type-checked then it is guaranteed to be free from errors.

Earlier work on type systems mainly focused on the results of a computation in a program in terms of its correctness. Then the study of type discipline and concurrency theory inspired the development of formal type systems that can statically formulate and verify the intended

properties of a program behavior, along with the permissible results of the computations.

Behavioral Type

Behavioral type theory is one such formalism that was originally proposed to verify concurrent programs based on process algebra (Nielson and Nielson, 1993, 1996). A behavioural type system uses behavioral types, a type-based abstraction, to formally describe the software entities such as communication protocols, interfaces, web services and contracts, as a sequence of operations in a concurrent and distributed environment. Formally speaking, a behavioral type system is a compositional type system that can directly model the interaction between system components as a notion of choice, causality and resource usage. Session types and behavioural contracts are two notions related to behavioural types.

Since the introduction of behavioral types, many type-based effect and proof systems, using the concepts of behavioural types, session types, spatial logic and processes as types, have been developed to study various correctness and behavioral properties such as unique receptiveness, race freedom, deadlocks, livelocks in large-scale concurrent and distributed systems (Sangiorgi, 1999; Chaki et al., 2002; Igarashi and Kobayashi, 2005; Dezani-Ciancaglini et al., 2005; Kobayashi and Sangiorgi, 2010). However, in these approaches, type-based specifications are explicitly added to the model and verify the usage patterns of resources and communication objects.

Recent work on behavioral separation in a distributed environment can be attributed to Caires (2008) who developed a spatial-behavioural typing system, to model the resource independence and synchronization in a distributed (concurrent) environment, based on the Spatial logic (Caires and Cardelli, 2003, 2002). The type system using parallel and sequential composition operators and resource ownership are handled using the type modality. Later, Caires and Seco (2013) developed a behavioral separation programming language based on λ-calculus to ensure the disciplined interference between resources in higher-order concurrent programs having fork/join parallelism. The language is based on the behavioral type systems (Honda et al., 1998; Chaki et al., 2002) that incorporates a behavioral view of the program properties and employs Concurrent Separation Logic (Reynolds, 2002; OHearn, 2007), to separate the dynamic behavior of run-time values rather than separating program states itself. However, program effects are computed based on the explicitly specified assertions to ensure the safety of concurrent programs.

Session Type

Session type, a notion of behavioral types, was originally introduced by Honda et al. (1998) to ensure the disciplined interaction between two partners in a distributed environment and later extended in the work of Honda (Honda et al., 2008) to incorporate the arbitrary number of participants in the same environment.

Recently, session types were extended with formal type system to control aliasing and to enforce usage protocols in concurrent environment for Java and Java-like languages, such as SessionJ (Hu et al., 2008, 2010), Yak (Milito and Caires, 2009) and Mungo (Gay et al., 2015a),

to name a few. Further, the linearity of resources is an important and recurring theme in concurrency that studies behavioral type systems, for process calculi and as mentioned by Honda (1993), Linear Logic can be considered as a source of inspiration for some aspects of session types.

Recent work on Linear Logic-based session types includes the work of Caires and Pfenning (2010), who proposed a Curry-Howard style interpretation of binary session types in intuitionistic Linear Logic, to expose a deep relationship between both concepts. Recently, with this line of work, Gay and Vasconcelos (2010) and Wadler (2014) developed a new calculus CP and a linear functional language GV, to establish a connection between session types and the Linear Logic, that can yield a process calculus, free from data races and deadlocks.

Typestate

Yet another important formalism, a notion of behavioural types, is typestate. Typestate was first defined by Strom and Yemini (1986) as a new programming language concept that determines the operations permitted on objects in a given context. According to Garcia et al. (2014) "typestate reflects how legal operations on objects can change at run-time as their internal state changes". Typestate associates state information with a variable of a given type, which is then subsequently used to decide the valid operations to be called on an instance of that type. Typestate is suitable to represent resources that follow state transition systems that follow the 'open then close' semantics. For example, a database connection can only execute a database command if it is in the open state.

Typestates were originally developed for imperative languages without the notion of objects, but later extended with the behavioural-type discipline, to support verification of object-oriented languages such as Vault (DeLine and Fähndrich, 2002), Fugue (DeLine and Fähndrich, 2004). Further, typestates were integrated as a first-class language construct in typestate-oriented language (Aldrich et al., 2009; Sunshine et al., 2011) to verify the correctness of the usage protocols in state-based sequential programs. In the new language, objects are being modeled not only as classes, but also the changing abstract states, where the correctness of the program is determined by tracking state transitions between different objects at execution time, thereby ensuring the correct usage of protocols.

Recent developments in state-based protocol checking and verification includes approaches (Caires and Seco, 2013; Garcia et al., 2014; Militão et al., 2014b; Gay et al., 2015b), that handle aliasing in a more robust way and verify communication protocols in distributed and concurrent object-oriented languages. These approaches ensure the basic memory safety conservatively, by associating typestate invariants with each referenced location and by ensuring that the type invariants hold for every store of this location. However, all the approaches discussed above focus on identifying violations of protocols but none of them check the higher-level (behavioral) characteristics of the protocols themselves, for example, their usage in practice and the complexity associated with their definition, that are vital in verifying correctness of many program properties (Beckman et al., 2011).

Ownership Type

Another stream of type-based systems is ownership type, a mechanism to express the sharing of program references (Noble et al., 1998; Clarke et al., 2013), in the way that allows controlled aliasing between objects by mitigating the undesirable effects to other objects.

Since its introduction, ownership types have been used in many formal approaches to provide safe aliasing control mechanisms (Boyapati and Rinard, 2001; Clarke and Wrigstad, 2003; Müller and Rudich, 2007; Cohen et al., 2009), and to control object deallocation explicitly (Matsakis et al., 2014). However, in ownership-based verification approaches, programmers define ownership invariants as locking or access information, to identify dependencies at the object level and use this information to avoid data races.

Atomic Set

Atomic sets have been used to detect atomicity violation and to avoid data races in concurrent programs. An atomic set defines a set of memory locations that share some consistency property and needs to be updated atomically. The atomic set can be viewed as a generalization of Hoare's monitors (Hoare, 1974) to multiple objects. They can be better integrated into the Java language.

Earlier work using atomic sets dates back to Vaziri et al. (2006)'s data centric programming model that defines `atomic set serializability`. Atomic set serializability is a disciplined interference criterion to avoid the problematic interleaving scenarios in the shared-memory programs based on atomic sets. Vaziri extended his previous work to support multiple object interference (Vaziri et al., 2010). Subsequent work used atomic sets to detect atomicity violation statically (Kidd et al., 2011) and dynamically (Xu et al., 2005; Hammer et al., 2008; Lai et al., 2010). The trend is followed by many local data-centric concurrency control mechanism and type systems (Dolby et al., 2012; Marino et al., 2013), to verify program behavior based on atomic sets.

Data-centric concurrency control is one alternative to the explicit locking mechanism. In contrast to the control-centric synchronization approaches, (Artho et al., 2003; Lu et al., 2008), where each program instruction is protected by synchronization constraints and then changes to the program states are tracked for every execution path of the program flow. The local data-centric approaches combine all fields of an object that require consistency, for all the control flow paths of program execution, into an atomic set and updates them atomically to avoid data races.

Contrary to data-centric concurrency control and the use of atomic sets, the development of type systems (Flanagan et al., 2003; Abadi et al., 2006; Flanagan et al., 2008) has been influential to ensure the atomicity and data-race freedom in concurrent programs, and to reduce the annotation overhead associated with manually adding synchronization primitives at code level. However, unlike atomic sets, in type systems, programmers provide explicit synchronization primitives, as locking and guarded specifications at field or class level, required by the code.

Recently, the notion of atomic sets was replaced by atomic variables by Paulino et al.

(2016). The objective was to handle the complexity associated with the use of memory structures in atomic sets. The proposed approach applies a resource-centered view of the data-centric concurrency control. However, in the proposed type system, programmers explicitly define the synchronization primitives on the individual data items that require atomic updates, to guarantee the progress of synchronization for all program execution scenarios.

Uniqueness and Immutability

Another area of interest has been the controlled sharing and interference of object references in imperative object-oriented programs. Sharing is a situation when a piece of memory is accessed by more than one reference, say `x` and y, so that a change to `x` affects `y` as well. Therefore, changes to one object may leave other objects in an inconsistent state, causing unwanted interference and subsequently, data races.

Work has been done to restrict the usage of references notably using access-based type annotations such as `uniqueness`, `immutability` and `read-only` (Clarke and Wrigstad, 2003; Boyland, 2006, 2010; Gordon et al., 2012; Clebsch et al., 2015). The objective was to identify isolated states that can be safely handled by one or more threads, thereby avoiding the unwanted interference and alternatively data races. However, the type system infers sharing effects such as `uniqueness` and `immutability` for the object references by computing an equivalence relationship for a set of free variables by evaluating the input expression. The inferred effects are then used to determine which part of the code can be safely shared between multiple threads to maintain the integrity of the data.

Recent development in this area is the Giannini's type and effect system (Giannini et al., 2018a,b) that expresses sharing in imperative programs based on the `pure` calculus (Capriccioli et al., 2016), where memory stores are modeled by rewriting the source code terms rather than by modifying the auxiliary storage.

Rely-guarantee Protocols

The logic-based program verification that employs rely-guarantee reasoning has been another active area of research that verifies the correctness of usage protocols and avoids inter-thread interference in concurrent programs (Parkinson and Bierman, 2005; Vafeiadis and Parkinson, 2007; Dinsdale-Young et al., 2010). However, in these approaches, rely-guarantee specifications are explicitly added at the state or thread level to model and control the concurrent interactions safely.

The most recent is a sub-structural type system (Militão et al., 2014a) that uses `rely-gurantee` protocol abstraction to model the interfering interaction of aliases to the shared states. The type system then ensures that the aliases always get a determined value regardless of the potential changes made by the program context during interleaving. However, the system explicitly assigns a separate role to each alias with a "rely ⇒ guarantee" relation between aliases. Later, Militão et al. (2016) extended the approach and developed a composition procedure based on linear capabilities (Morrisett et al., 2005), to address the decidability of protocol composition and its integration with the protocol abstraction.

Separation Logic

Among other logic-based approaches for data race freedom, separation logic that is based on Hoare logic, attained much attention for controlling aliasing and verifying program behavior.

Hoare's logic for conditional critical regions (Hoare, 1972) and monitors (Hoare, 1974) was widely adopted because of the simplicity and practicality of their use in Java-like programs. Hoare's logic limits thread interference to a few synchronization points. However, it cannot syntactically enforce a safe monitor synchronization. This is because of the potential risk of aliasing in a Java program where multiple threads can manipulate shared-memory data in an unsafe manner. O'Hearn (O'Hearn et al., 2001) and Reynolds (Reynolds, 2002) extended the Hoare's logic to Classic Separation Logic, a new program logic with new connectives and separation conjunction ($*$), to reason about the sequential programs that manipulates pointer data structures.

Hoare logic uses triples $\{P\}S\{Q\}$ where P and Q are predicates over program states that define the `required` and `ensured` properties of an expression statement S. However, in Separation Logic, the idea is to explicitly divide each program state, related to a current method call, into a heap and a store part, to allow explicit local reasoning about the heap memory. In this approach, the heap `h` is divide into two disjoint parts say h_1 and h_2 using separation formula of the form $\phi_1 * \phi_2$ where ϕ_1 is a pointer valid for the part h_1 and ϕ_2 is a pointer valid for the part h_2. The conjunction $*$ operator combines two disjoint parts of the same heap. The idea of using separation formula to verify heap structure, dates back to the seminal work of Reynolds (Reynolds, 1978) who proposed a syntactic interference control mechanism to constrain the effects of interference in Algol-like languages using the Separation Logic. The separation formula ensures that two threads accessing the same location do not interfere to verify program behavior

Eventually, Separation Logic was realized as a new program logic called Concurrent Separation Logic (CSL) (Brookes, 2004; OHearn, 2007) to reason about multi-threaded programs, with an assumption that if two threads can operate on disjoint parts of the same heap location without interfering with each other, they can be verified in a safe and isolated way. CSL enforces correct synchronization of the shared-memory data logically, rather than syntactically. The idea of CSL was then extended in several sub-structural type systems and concurrent approaches (Gotsman et al., 2007; Appel and Blazy, 2007; Hobor et al., 2008). to guarantee data race freedom in the shared-memory concurrent programs, and have recently been applied in high-order imperative concurrent languages and type systems (Schwinghammer et al., 2011; Jensen and Birkedal, 2012). However, in the separation logics-based approaches, the separation predicates are explicitly specified in the program to define the access rights on the memory locations.

A Move to Permission-based Specifications

Among all formal approaches to the verification of shared-memory programs such as atomic set, behavioral type, session-type, relay-guarantee reasoning and Separation Logic, is access permission. Access permission is an abstract capability that combines type (effect) systems

and, provides more advanced support for reasoning about the heap resources (Boyland, 2003).

The notion of access permissions is built on Linear Logic (Girard, 1987), that treats permissions as linear resources, and the Separation Logic (O'Hearn et al., 2001) that performs local reasoning of program behavior against specifications. However, Separation Logic does not support the concurrent read access of a memory location by multiple references or threads. Therefore, Boyland (2003) and Bornat et al. (2005) combined separation logic with abstract capabilities, called access permissions, to allow concurrent reading of a program state.

Compared to classic verification methods such as Owicki-Gries (Owicki and Gries, 1976) for concurrent programs, the permission-based Separation Logic ensures that: a) only one reference (thread) can write on a particular location at any given time, thereby mutating data in a safe way; b) if a location is read by a thread, all other threads can only have read permission for that location, thereby implying data race freedom without the need to explicitly check for the interference between threads in concurrent programs.

Plural (Bierhoff and Aldrich, 2007; Beckman et al., 2008), a permission-based program verifier, was the next development phase where access permissions were combined with typestate abstractions to statically ensure the mutability of object's states, following the separation logic, and to verify protocol compliance in Java-like sequential and concurrent programs.

Access permission was then subsequently incorporated as first class language constructs in permission-based programming paradigms (Stork et al., 2009; Aldrich et al., 2011), to parallelize execution of sequential programs in a safe way.

6.2 Permission-based Verification of API Protocols

This section introduces the first dimension of the state of the art permission-based verification approaches in detail. The focus is on the verification of API protocols for single- and multi-threaded programs. Table 6.1 provides a summary of the permission-based protocol verification approaches studied in this research.

6.2.1 Verification of API protocols in Single-threaded Programs

In object-oriented programs, objects define usage protocols. `Usage protocols` are constraints on the order of method invocations that the client of the protocol must follow for the underlying objects to work properly. Typestates have been used to specify usage protocols in many formal approaches where state information is associated with a variable of a given type, which is subsequently tracked to decide the valid operation sequence to be called on an instance of that type. It is generally acknowledged that statically tracking object's state is a challenging task in the presence of unrestricted aliasing. This can be attributed to the improper and unexpected state transition during program execution, and that can happen in situations such as a) when a method uses a data structure having aliases deeply nested in the hierarchy and causes side effects, or b) when aliased parameters are passed to a method expecting non-aliased parameters.

Table 6.1: Access permission-based protocol verification.

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Bierhoff and Aldrich (2007)	Seq	Plural (NSL)	Plural	(St,D)	Sym	(U,S,F,P,I)	N	Y	protocol verification
Beckman et al. (2008)	Con	Plural (NSL)	Plural	St	Sym	(U,S,F,P,I)	N	Y	protocol verification, race conditions with atomic blocks
Beckman (2009)	Con	Plural (NSL)	Sync-or-Swim	St	Sym	(U,I,S,F,P)	N	Y	protocol verification race conditions with synchronized blocks
Militao et al. (2010)	Seq	-	Plural	St	Sym	(U,I,S,F,P)	N	Y	race conditions
Bierhoff (2011)	Seq	Java-like (NSL)	JavaSyp	St	Sym	(U,I)	N	Y	CME exceptions
Aldrich et al. (2011) Aldrich et al. (2012)	Seq	Plaid (NSL)	Plaid	(St, D)	Sym	(U,S,I)	N	Y	protocol verification
Naden et al. (2012)	Seq& Con	Plaid (NSL)	Plaid	(St,D)	Sym	(U,S,I)	N	Y	race conditions

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, *Sym* = symbolic, Fract = fractional, U = unique, I = immutable, S = share, F = full, P = pure, NSL = new specification language, $\mathbb{Z}$ set of Integers, $\mathbb{R}$ set of Real numbers, $\mathbb{N}$ set of positive Integers.

Access permissions provide a flexible aliasing control mechanisms to track all the references of a particular object and update state changes to all such references. Therefore, access permissions have been used, as part of formal specifications, to specify the intended design and to verify the correctness of typestate-based usage protocols in many formal approaches such as Plural (Bierhoff and Aldrich, 2007, 2008; Bierhoff, Kevin, 2009; Bierhoff et al., 2009b), JavaSyp (Bierhoff, 2011) and Plaid (Aldrich et al., 2011, 2012) and other related techniques.

Typestate Verification using Linear Logic

Bierhoff, Kevin (2009) presented a formal specification language and a type system to soundly and modularly verify API protocols in sequential programs based on access permissions. The aim was twofold, firstly to verify protocol conformance with actual program implementation in the presence of aliasing, and secondly to check whether the client of the program obeys the specified protocol.

With this stream of work, Bierhoff (Bierhoff and Aldrich, 2007) presented a permission-based modular protocol checking approach for a Java-like object-oriented language.

In this approach, programmers express their design intents, as a valid sequence of events associated with a particular object, and the aliasing information using permission-based typestate contracts, written in Linear logic-based specifications at the method level. The system supports five kinds of symbolic permissions i.e., unique, immutable, full, share and pure. The Linear logic operators are already explained in Section 2.1.1, except the additive disjunction ($\oplus$) that represents an alternative occurrence of multiple tasks.

Listing 6.1 shows a sample permissions-based typestate contract for a read-only iterator. The aim is to avoid the concurrent modification of `Collection` object when the `iterator` is in progress.

Listing 6.1: Permission-based typestate specifications of a read-only Iterator (Bierhoff and Aldrich, 2007).

```
1 interface Iterator<c : Collection, k : Fract> {
2 states available, end alive
3 boolean hasNext():
4 pure(this) ⊸ (result = true ⊗ pure(this) in available)
5            ⊕ (result = false ⊗ pure(this) in end)
6 Object next(): full(this) in available ⊸ full(this)
7 // other methods such as finalize() etc. can be written similarly
8 interface Collection {
9  //  methods such as  add(), size(), remove(), contains() etc. comes here
10  Iterator<this, k > iterator(): immutable(this, k)⊸ unique(result)}
11 }
```

According to the usage protocol, an `iterator` can be in one of the two states at any moment i.e. `available` or `end`. The state `alive` is a state inherited from the root `Object` type (Line 2). In Line 11, an `iterator` object is created with unique permission in `Collection` class. Importantly, it can be observed that when an iterator is created, it stores a reference to a collection object being iterated in one of its fields. This reference should be associated with the appropriate permission i.e., immutable to guarantee immutability of the `Collection` object while iteration is in progress.

The pre-condition (pure(this)) in Line 4 specifies that method `hasNext()` requires pure permission on the referenced object as it just tests or reads iterator's state. As method `next()` can change `iterator`'s state, it needs full (this) (Line 6). The method `hasNext()` determines whether another object is already present in the Collection object with `available` state, or if the iteration has reached its `end`. The post-condition (Line 4 and 5) specifies if the result is true. It is legal to call the `next()` method on the same object in the `available` state, Otherwise, it is illegal. The post-conditions of both methods further show that they return the consumed permission back on the referenced object when they exit. The system leverages this information through method implementations to track state transition in the presence of aliasing, to guarantee the absence of concurrent modifications, and to verify whether program implementation follows the design intents.

This approach is realised in Plural (Bierhoff and Aldrich, 2008), a permission-based automated protocol checking and conformance tool, implemented as a Java Eclipse plugin. In Plural, a program is specified with permission-based pre and post-conditions at method (parameter) level using JSR-175* annotations to check whether the client of the APIs follows the specified protocol.

Plural performs intra-procedural static analysis, called DFA (Data Flow Analysis) of the annotated code to identify and track specified pre- and post-permissions across method calls for every program variable (parameter, receiver object, and local variable) and issues warning for protocol violations in the program. As part of the analysis, it implements a Crystal analyzer that performs dynamic state tests (branch-sensitive flow analysis) to track and report exceptions to the underlying objects. It further checks the structure of the provided specifications by implementing an Annotation analyzer. The Effect checker in Plural identifies whether a method

*https://jcp.org/en/jsr/detail?id=175

is immutable or whether it produces side effects. The Fraction analyser tracks the flow of permissions through the system to split and join permissions associated with a referenced object.

Later, Bierhoff et al. (2009b) extended the modular protocol checking approach to explore the soundness and effectiveness of Plural in specifying large case studies for real APIs and large third party open-source code bases, For example, Database Connectivity (JDBC) API in Apache Beehive project[†] and PMD, a static code analyzer from DaCapo benchmark[‡] that implements Java `iterator` API. The objective is to measure the precision in terms of false positives, the computational cost and the annotation overhead associated in manually specifying and verifying these APIs with Plural annotations.

The approach follows the Design by Contract principle to explicitly specify state invariants at the method level. State invariants are permission-based typestate assertions with a valid typestate that should hold when an object is in a specific state. The approach uses the concepts of 'capture' and 'release' permissions to avoid inter-object dependencies at the method level. Listing 6.2 shows a sample code snippet in Plural for `Connection` class in Java JDBC API.

Listing 6.2: A Java JDBC connection interface (fragment) with Plural specifications (Bierhoff et al., 2009b).

```
1 @States({"open", "closed"})
2 public interface Connection {
3  @Capture(param = "conn")
4  @Perm(requires = "share(this, open)", ensures = "unique(result) in open")
5  Statement createStatement() throws SQLException;
6  @Full(ensures = "closed")
7  void close() throws SQLException;
8 }
```

In Listing 6.2, `@States` annotation (Line 1) specifies concurrent typestates for a `connection` object. The method `createStatement()` (Line 5) creates statements, in the `Connection` interface, with unique permission in 'open' state. When connection object is closed it invalidates all the statements created with it, leading to runtime errors if a programmer uses invalidated statements. To avoid this error, the approach captures the dependent `conn` object, using `@Capture` annotation in Line 3. The annotation @Perm in Line 4 with `requires`, clause specifies share permission on the captured object as pre-permission with `open` state guaranteed, and the `ensures` clause specifies that the method returns a new statement object in open state with unique permission on it.

The permissions on the `conn` object are explicitly released (using `@Release`) when the statement object is no longer in use or when the connection object is in `closed` state. The method `close()` closes the `conn` object by calling method `isClosed()`, (not included in the sample program due to brevity), that ensures the current state of the connection object to be `closed` (using annotation @TrueIndicates ("closed")). The ensures clause in Line 6 specifies that method returns Full permission back to the connection object in the `closed` state. The analysis tracks these specifications in the system to verify protocol compliance with program implementation and to verify correct usage of the protocol by the client program.

[†] http://beehive.apache.org/
[‡] http://dacapobench.org/

Typestate Verification using Plaid

Aldrich et al. (2011, 2012) developed a new permission-based programming language called Plaid. The aim was to extend the previous typestate-oriented language (Aldrich et al., 2009) with first class-states and access permissions. Plaid was originally designed to track the typestate of a referenced object at runtime and to handle unrestricted aliasing using permission-based typestate information. Typestate-oriented programming supports typestate as part of the language where typestate of an object is directly associated with its class (type) and that class can change dynamically.

Every type in Plaid is represented as tuples having a type structure and associated permissions that express the aliasing and the mutability of the corresponding object's typestate. Plaid borrows its grammar and lexical structure from the Java Specification Language (JSL) and provides interoperability with Java programs. Classes in Plaid are represented using the keyword `state` and transitions between states are represented by the state transition symbol '≫' that distinguishes pre-state from post-state. Plaid supports three types of symbolic access permissions i.e., unique, immutable and share in the specifications. The keyword 'none' is used when no permissions are required to access an object.

Listing 6.3 shows an annotated code fragment for a buffer implementation in Plaid. A buffer can be in one of the two states i.e. `EmptyBuffer` and `FullBuffer` (Line 1). The method `put()` is associated with an empty buffer. The signature of the method `put()` (Line 4) specifies that the state of the receiver object should change from `EmptyBuffer` to `FullBuffer` when the buffer receives some element. The state `FullBuffer` requires a field element `elem` that is passed as method parameter `e`. The permission-based contract (Line 4) specifies that the passed element has unique permission in the `Element` state and the method does not return any permission (`none`) to the caller of the method. This is because a field reference with exclusive rights (unique) has been created for that element in `FullBuffer` state. Otherwise returning permission back to a caller would cause a violation of the uniqueness property. Likewise, the `FullBuffer` state has a single operation `get()` (not included here due to brevity), that returns the current state of the object in reference `elem` and ensures that the receiver object will go back to an `EmptyBuffer` state.

Listing 6.3: A buffer implementation (fragment) in Plaid (Aldrich et al., 2011).

```
1 state Buffer comprises EmptyBuffer, FullBuffer {}
2 ...
3  state EmptyBuffer caseof Buffer {
4  method void put(unique Element ≫ none e) [EmptyBuffer ≫ FullBuffer] {
5    this ← FullBuffer {elem = e};}
6 }
```

Plaid runtime leverages permission flow through the system along with associated typestate information to ensure protocol compliance at runtime. The type system allows permission splitting, joining and type casting automatically (when and where required) using the permission splitting and joining rules given in Table 2.2.

Naden et al. (2012) presented a type system and a flexible permission borrowing mechanism for `unique`, `shared`, and `immutable` permissions without using explicit fractions of permissions. Permission borrowing is an extraction of permission from a source field,

temporarily using the borrowed permission, and returning part or all of it to the source field. The aim was to prevent the concurrent modifications of the shared objects based on symbolic permissions.

The type system is based on the Plaid language. Like Plaid, it supports three types of symbolic permissions i.e unique, immutable and share but Unlike Plaid that has weak permission borrowing support and that reassign a field value to recover permission on it, and where references are threaded explicitly from one call to an other, Naden's system binds access permissions with a reference and this permission is consumed once the reference is passed to a function. The caller function then returns the original permission to the same reference.

Unlike other techniques (Boyland, 2003; Bierhoff and Aldrich, 2007; Jacobs et al., 2011; Heule et al., 2011) that support permission borrowing, this approach provides a more intuitive and natural abstraction to model and to reason about the permission flow through the system, making permission tracking flexible and much easier for programmers. However, like Plaid, it wants programmers to explicitly specify permissions-based state information as a part of method specifications.

Typestate Verification using JML

Bierhoff (2011) combines symbolic permissions (Bierhoff and Aldrich, 2007), with JML contracts (Leavens and Cheon, 2006) to reason about aliasing and to detect the absence of Concurrent Modification Exceptions (CMEs) and other recurrent programming errors, such as `IndexOutofBoundsExceptions` exceptions in realistic data structures such as Java ArrayList.

Although JML specifications have been used to verify functional correctness and domain specific properties of sequential and concurrent programs (Rodríguez et al., 2005; Araujo et al., 2008; Kim et al., 2009; Cok, 2011), JML's support for concurrency and aliasing is rather limited. In contrast, access permission provides flexible aliasing control mechanism to track all the references of a particular object and update state changes to all such references. The presented approach defines permission-based class invariants as JML contracts.

The technique implements a permission tracking algorithm as a prototype tool called JavaSyp[§](Symbolic Permissions for efficient static program verification). In JavaSyp, permission-based invariants are specified using Java annotations and tracked as part of the type checking procedure, to ensure that the specified invariants hold as long as a the client has permission to the referenced object and to control aliasing. In this approach, permission tracking is straightforward as tracking symbolic values is much easier than tracking fractional permissions. However, it only supports two kinds of permissions, i.e., unique and immutable using annotations `@Excl` and `@Imm` respectively with the referenced object.

Listing 6.4 shows an annotated version of conventional Java ArrayList object `a` declared with unique permission in Line 2. The list object maintains a list of elements in the order placed originally. The method `getElem()` method returns the element on the given location (`index`) with immutable permission on it (Line 6). The invariants (Line 4) for method

[§]http://code.google.com/p/syper.

`getElem()` specifies that object `a` should be a non-null reference having at least one element in it and the total number of elements in `a` should not exceed the declared `size`. The annotation `@requires` in Line 5 specifies a pre-condition for method `getElem()` that, before calling this method, the `index` parameter must be between `0` and `size-1`. JavaSyp performs static analysis of the annotated code to generate verification conditions (VCs) based on the specifications. The program is then verified against inferred conditions using the SMT solver (C. Barrett A. Stump and Tinelli, 2010).

Listing 6.4: A Java array list (fragment) with permission-based JML contract in JavaSyp (Bierhoff, 2011).

```
1 public class ArrayList<T> {
2  @Excl private T[] a;
3  private int size ;
4  //@invariant 0 <= size & a != null & size <= a.length;
5  //@requires  0 <= index & index < size ;
6  @Imm public T getElem(int index) {
7    imm: return a[index];}
8 }
```

6.2.2 Verification of API Protocols in Multi-threaded Programs

Beckman (2010) extended the Bieroff's modular automatic protocol checking approach to verify "if the object protocols work correctly even in the presence of concurrent modifications by multiple threads". This section discusses the use of permission-based specifications to verify usage protocols in concurrent programs (Beckman et al., 2008; Beckman, 2009; Militao et al., 2010) using different mechanisms.

Typestate Verification with Atomic blocks

Beckman et al. (2008) extended the permission-based modular protocol checking approach (Bierhoff and Aldrich, 2007) to verify the correctness of usage protocols for a set of concurrent programs such as JChannel[¶] and Reservation Manager that use atomic blocks as synchronization primitives. The objective was to enforce the correct use of typestates at runtime and to verify API protocol compliance with its specifications. The approach uses five kinds of symbolic permissions to identify aliasing and to approximate whether a referenced object can be thread-shared or not.

Listing 6.5 shows a sample method `isConnected()` in a `Connection` class with permission-based typestate specifications.

Listing 6.5: Permission-based typestate specifications for method isConnected() in a Connection class (Beckman et al., 2008).

```
1 class Connection {
2 boolean isConnected():share(this, ?) ⊸
3  (result == true ⊗ share(this, CONNECTED)) ⊕
4  (result == false ⊗ share(this, IDLE)) {
5  atomic:{
6   return (this.socket != null);}
7  }
8 }
```

[¶] http://www.cs.cmu.edu/~nbeckman/research/atomicver/

The pre-condition "`share(this, ?)`" in Line 2 asserts that the method needs share permission on the receiver object, which needs to be in the unknown (`?`) state. Likewise, the post-condition in Line 3 specifies that if the method returns `true`, the receiver object should get the original permission back while in the `CONNECTED` state Otherwise, it would get the same permission back but in the `IDLE` state in Line 4. Exclusive access to full, share and pure references are maintained using atomic blocks in Line 5.

The approach is realised as part of the Plural tool. The analysis identifies the abstract state of a referenced object before calling a method, and discovers the way it would be shared with other objects. If the permission on a particular reference (thread) indicates that the referenced object can be accessed simultaneously by other references (threads), as is the case with full, share and pure permissions, it assumes that the object is thread-shared.

The analysis discards state information of the local variables having pure and share permissions as the objects with these permissions can be modified by other threads and it is difficult to track them statically through atomic blocks. The limitation of this work is the use of atomic blocks as mutual exclusion primitives. Atomic blocks are associated with transactional memory systems and have limited usage in today's applications. The current analysis generates false positives for programs having synchronization primitives other than atomic blocks.

Typestate Verification with Synchronized blocks

Beckman (2009) extended the previous type system to perform typestate verification of concurrent programs having synchronized blocks as mutual exclusion primitives.

In this approach, every program reference is associated with a permission kind (having a permission type and an abstract state that is part of the reference type). Like Beckman et al. (2008), the system distinguishes between thread-local and thread-shared objects based on permission contracts. The approach is implemented in a tool called Sync-or-Swim for Java that performs static analysis of the program (within a method). It identifies the references on which the current thread is known to have synchronized and tracks the permissions associated with references as they flow with method's pre- and post-conditions, to ensure that an object can be modified concurrently. The analysis discards state information for thread-shared (modified by other references) objects unless it is statically known that the same references have previously been synchronized.

Unlike other behavioral checkers for concurrent programs (Jacobs et al., 2005) that requires lock-based specifications to identify the part of the heap to be protected, the proposed approach can verify program behavior without requiring lock-based specifications. Like other single-threaded typestate verification approaches, it only requires aliasing (permission) and typestate information to verify the correctness of concurrent programs.

Typestate Verification with Views

Militao et al. (2010) expands on Bierhoff's permission type system for Plural (Bierhoff and Aldrich, 2007), by going beyond the five traditional types of permissions. The approach introduces a new abstraction called `View` that is a projection of an object with a small set

of access permissions associated with individual components (fields and/or methods) of an object.

The type system combines view-based controlled aliasing with typestates and Boyland's fractional permissions to manage safe initialization of different sections of an object reference, to track state information and to ensure safe access of the referenced objects in a unique-writer and multiple-readers scenario. An `immutable` view can be shared with an inbound number of copies and a `write` request merges all the readers back to a single writer using fractional permissions. However, it does not support aliasing of the form where an object can be shared between multiple writers with a state guarantee. In this approach, a `view` behaves as a state except that it can be split, merged (recombined) using fractional permissions. Therefore, it resembles the permission accounting model (Bornat et al., 2005) where `views` are treated as accountable `parts` of a typestate thereby, allowing local reasoning of the shared-memory programs.

The analysis of all the Plural-based verification approaches in Section 6.2.1 and 6.2.2 shows that Plural can identify common challenges for specifying and implementing usage protocols in real-world case studies. It helps programmers to statically follow usage protocols without actually executing the program.

However, Plural analysis is limited as it cannot identify errors in the specifications and might use misspelled (non-existent) specifications. Moreover, there is no reachability analysis support in Plural, which means that a programmer may write wrong specifications at the method or class level and methods with these specifications will never be called by any client code, resulting in unreachable (dead) states at the method level. Complementing the Plural tool, research (Siminiceanu et al., 2012), has been done to verify the correctness of manually added Plural specifications as well as verifying the program behavior. Further, it provides limited support to the verification of typestate invariants. It can only check invariants on boolean properties, e.g. checking for `non-nullness`, However, it cannot verify invariants that involve arithmetic predicates, e.g. $x > 0$ (where x is an integer field). Moreover, it requires programmers to explicitly specify the design intents of the API protocols as permission-based typestate specifications in the program which results in annotation overhead for the programmers.

In the same fashion, Plaid and related approaches forces a client program to follow the desired sequence of method calls to verify the correctness of typestate-based programs, but at the cost of adding permission-based annotations as a part of type declarations at the code level. Moreover, Plaid does not support `full` and `pure` permission as for its analysis. In Plaid, permission-based typestate specifications are added as a part of the program to verify proper state transition between multiple objects during execution of a method.

6.3 Verification of Race conditions and Deadlocks in Multi-threaded Programs

In imperative and object-oriented programming languages, the biggest challenge has been the correctness of multi-threaded programs in the presence of aliasing. Access permissions have been used to characterize the way a shared resource can be accessed by multiple threads and to handle aliasing in many verification approaches. The general idea is to assign permission to program references to access memory locations and track the permission flow through the system to enforce mutual exclusion mechanisms in shared-memory concurrent programs.

This section presents the second dimension of the review i.e., the use of permission-based specifications for verification of domain specific problems such as deadlocks and race conditions in multi-threaded programs.

6.3.1 Permission Sharing and Accounting Models

Permission sharing and accounting models (Boyland, 2003; Bornat et al., 2005; Parkinson and Bierman, 2005; Appel and Blazy, 2007; Hobor et al., 2008; Dockins et al., 2009) facilitate thread-local reasoning for shared-memory concurrent programs and to ensure race-free sharing of heap locations.

The Boyland's permission sharing model (Boyland, 2003) using fractional permissions, defines shared ownership of resources in a concurrent environment. The model supports rational number $\mathbb{R}$ in the range $[0, 1]$. The sharing policy maps permission quanta x to a value to allow operations on a particular memory location. For example, 1 represents full (read and write) ownership, $0 < x < 1$ represents read-only ownership while 0 means no ownership.

The fractional model has been used to handle problems that follow concurrent divide-and-conquer algorithms where a shared (read) permission can be divided into multiple shared permissions, to an unbounded depth, for any possible pattern of divide-and-conquer. Although fractional share model is infinitely splittable, it does not satisfy the disjointness property because rational numbers are not ideal for sharing, as shown by Parkinson and Bierman (2005), who proposed a permission sharing model using subsets of natural numbers $\mathbb{N}$ to formalize and verify single-threaded Java applications with separation logic. In this model, resource invariants are defined using abstract predicates defined at the `Object` class level with an empty footprint, (permissions associated with a memory location), that each subclass extends to hold additional fields.

Bornat et al. (2005) proposed a permission accounting model and a light-weight verification approach to handle the accounting problem associated with reader-writer locks in concurrent programs. The approach extends separation relation $\mapsto$ in classic Separation Logic (Reynolds, 2002) and associates fractional permissions with each heap location to allow read sharing of heap locations. In this approach, each heap location x is treated as a map having addresses E with a permission value z, where z represents the level of permission carried by a heap location, as shown in Formula 6.1).

$$x \underset{z}{\mapsto} E \Rightarrow 0 \leq z \leq 1 \tag{6.1}$$

The idea is to count the number of shared tokens using an integer counter say s. It is incremented or decremented when a reader locks (receives a read token) or unlocks (returns the share token) respectively, $s > 0$ means there are outstanding read tokens but $s = 0$ implies the absence of outstanding readers, which means that a writer may acquire, hence ensuring a race-free sharing of heap locations.

Appel and Blazy (2007) presented the operational semantics and developed a Sequential Separation Logic to extend the C Minor language, a mid-level imperative programming language as a machine independent-intermediate language. The approach evaluates expressions as functions in Coq and provides an end-to-end machine-checked soundness proof of the proposed logic in Coq (Leroy, 2006). Coq, a part of CompCert project[‖], is a proof-correct optimization compiler from a high-level intermediate language such as C Minor to assembly language for Power PC architecture. Unlike the classical Separation Logic (O'Hearn et al., 2001) where expressions are evaluated independent of heaps, the approach associates each expression evaluation with a footprint.

"A footprint is a mapping from memory addresses (ν) to permissions" (Appel and Blazy, 2007). In the proposed semantics, footprint (ϕ) is considered as a set of fractional permissions (Bornat et al., 2005) to verify non-interference of load and store operations in memory. A memory store yields result only if reading or writing a chunk of memory type, say ch at location ν is legal according to its footprint. For example, the semantics $\phi \vdash load_{ch}\nu$ (or $\phi \vdash store_{ch}\nu$) depicts that all the addresses from location ν to $\nu + |ch| - 1$ can be read (or write). Loading memory outside the footprint yields exceptions and causes expression evaluation to stop. The disjoint sum of two footprints $\phi_0 \oplus \phi_1 = \phi$ ensures the exclusive read/write or read-only ownership of the underlying memory.

Hobor et al. (2008) extended the Appel and Blazy's machine-checked soundness proof and Leroy's compiler-correctness proof in a concurrent setting, and developed a concurrent C Minor language having shared-memory and first-class locks and thread. He proposed a modular concurrent operation semantics as a generalization of Concurrent Separation Logic (CSL) (OHearn, 2007) but it goes beyond CSL as it allows dynamic lock and thread creation.

In the semantics, a world w corresponds to a footprint ϕ as in the work of Appel and Blazy (Appel and Blazy, 2007). A world specifies permissions for the current thread but this semantic deals with load (store) operations for multiple threads. The need was to evaluate an expression with a guarantee that footprints (ϕ) of different threads are disjoint. For this purpose, the approach defines permission-based lock invariants to grant or restrict ownership for the accessed memory by extending the classic separation relation $\mapsto$ as follows:

$$e \underset{\pi}{\mapsto} R \tag{6.2}$$

[‖] http://coq.inria.fr

The relation shows an expression e maps to a memory address with resource invariant R. Every expression acquires a lock before its evaluation. Each lock is associated with a resource invariant R, where each invariant is supported by a unique set of memory addresses and worlds that inform the lock ownership π acquired or lost by each thread. The approach implements Parkinson's (Parkinson and Bierman, 2005) permission sharing model to define ownership. A 100% share represents full ownership and a non-empty ownership ($0 < \pi <$ 100%) represents read-only access. Any access without ownership means the program has no semantics and the evaluation stops.

Dockins et al. (2009) proposed a tree share permission accounting model that is more powerful than Bornat's token accounting model. It rectifies the problems with Parkinson's permission model. Tree share is a boolean-labelled binary tree that supports both splitting and token accounting for the shared reading of resources in concurrent settings. Although Boyland's fractional share model is infinitely splittable, it does not satisfy the disjointness property and may pose read/write and write/write race conditions. Similarly, in Bornat's token accounting model, a central authority lends out the total permission into shares in the form of permission tokens when and where required. It counts the outstanding tokens to verify permission accounting. These models satisfy positivity of resources but not the disjointedness.

Unlike previous share accounting models, Dockins et al. (2009) defines heaps as partial functions from memory locations (L) to values (V) with pairs of non-unit shares (S). The token factories are represented using non-negative integers and the tokens themselves are represented using negative integers. When a token is pushed back into the factory, the integers are added. The token factory's share becomes zero when it gets all its tokens back. The extended points-to operator relation is given below:

$$l \underset{\text{s, n}}{\mapsto} v \tag{6.3}$$

The relation specifies that memory location l contains a value v with a non-unit share s that is indexed by an integer n. If n is zero, the share s is full. If n is positive, it means that n tokens are missing over s in the token factory, and the negative value for n depicts that token factory has $size - n$ shares. The extended model supports and satisfies all the required properties such as disjointedness, cross, and infinite splitting in permission sharing models.

6.3.2 Permission-based Verification Techniques and Tools

This section describes and discusses the permission-based verification approaches and tools such as Fluid (Zhao, 2007), Chalice (Leino et al., 2009; Leino and Müller, 2009), Verifast(Jacobs et al., 2010, 2011), Pulse(Siminiceanu et al., 2012; Cataño et al., 2014; Ahmed and Cataño, 2018), Heap-Hop (Villard et al., 2010), HIP/SLEEK (Hobor and Gherghina, 2012; Jacobs and Piessens, 2011), HJp (Westbrook et al., 2012), Vercors (Amighi et al., 2012, 2014; Huisman and Mostowski, 2015; Amighi et al., 2015) and Viper (Juhasz et al., 2014; Müller et al., 2017) that have been developed to resolve concurrency problems in concurrent programs.

Table 6.2 provides a summary of the existing permission-based verifications tools and related approaches to verify common concurrency problems.

Fluid

Zhao (2007) developed a permission-based language and a type system to enforce fixed locking order mechanism in Java-style multi-threaded programs having unstructured parallelism and synchronization.

The technique was realised as a prototype tool in the Fluid project (Greenhouse and Scherlis, 2002; Greenhouse, 2003). In their approach, a program is explicitly annotated with a method's effects and lock invariants. The method's effect specifies the read or write operation on the current object `this` or any of its field, e.g. `reads(this.x)` and `writes(this.x)` in Listing 6.6, in Line 2 and 4 respectively. The lock invariant specifies the synchronized access of a referenced object inside the method body using the `requires (this)` clause in Line 7, that a method call for `deposit()` should be inside a synchronized block to acquire a `lock` on the receiver object `this`.

Listing 6.6: Code segments showing read, write and lock usage annotations in Fluid.

```
1 class Account{
2  read (this.x)
3  void getBalance(){ x; }
4  writes (this.x)
5  void setBalance(int newX) { x = newX; }
6  void deposit(int x){
7  requires (this) { balance = balance + x; }
8 }
```

The system then translates high level access annotations into low-level (fractional) permissions to distinguish the read and write effects of a method on the referenced objects. The system assigns unique permission with a referenced object if it is being written in the method body and a value less than 1 is assigned for read operation. The type system uses this information to ensure that a given expression can be executed with assigned permissions but it does not verify program behavior based on input specifications.

Further, Zhao et al. (2008) proposed a synchronization policy to avoid the unnecessary synchronization effects in the previous approach. The system uses "permission nesting" to interpret the safe and correct usage of lock-based specifications associated with a field.

Chalice

Leino and Müller (2009) presented a permission-based verification method to prevent problems such as deadlocks and race conditions, that arise due to dynamic locking orders in multi-threaded programs. The approach ensures concurrent sharing and un-sharing of objects among multiple threads based on Boyland's fractional permissions (Boyland, 2003). The system uses permission percentages (between 0 and 100) instead of permission fractions. A permission percentages is a fractional permission with a definite size that splits a field permission among several monitors or threads. A thread can access a shared object, (heap

Table 6.2: Permission-based verification of race conditions and deadlocks.

Reference	Prog	Language	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Boyland (2003)	Con	PSM	-	-	Frac	[0, 1]∩ℝ	N	-	race conditions
Bornat et al. (2005)	Con	PSAM	-	-	Count	[0, 1]∩ℤ	N	-	race conditions
Parkinson and Bierman (2005)	Seq	Java-like (NSL)	-	-	Counting	total read*	N	-	modular reasoning about abstract datatypes and information hiding
Appel and Blazy (2007)	Seq	CMinor (PSAM)	Coq	-	Counting	[0,1]∩ℕ	N	-	race conditions and machine-checked correctness proof.
Zhao (2007) Zhao et al. (2008)	Con	Java	Fluid†	St	Frac	read write*	N	Y	race conditions
Hobor et al. (2008)	Con	CMinor (PSAM)	Coq	-	Counting	[0, 100]%	N	-	race conditions and machine-checked correctness proof of CSL.
Dockins et al. (2009)	Con	CMinor	Coq	Frac	-	(s, n) where n ∈ ℤ	-	Y	provides a strong CSL semantics that fully supports both permission splitting and token counting problems
Leino and Müller (2009)	Con	Chalice	Chalice	D	Frac	full some no	N	Y	race conditions and deadlock
Leino et al. (2009)	Con	Chalice	Chalice	D	Frac	acc(x) rd(x)‡	access pure	Y	race conditions and deadlock
Villard et al. (2010)	Con	NSL	Heap-Hop	SFEA	-	x ↦ C	N	Y	race conditions, deadlocks absence of memory leaks protocol verification protocols
Jacobs et al. (2010, 2011)	Seq& Con	C and Java-like (NSL)	VeriFast	D	Frac Count	read write*	N	Y	race conditions Divided by zero error, NullPointer and ArrayIndexOutOfBoundsExceptions
Jacobs and Piessens (2011)	Con	NSL	VeriFast	D	Frac Count	[0, 1]∩ℝ	N	Y	fine-grained synchronization mechanism to avoid race conditions
Hobor and Gherghina (2012)	Con	NSL	HIP/SLEEK	D	Frac	(0, 1]∩ℝ	N	Y	race condition using barriers in Pthread library
Westbrook et al. (2012)	Con	HJ	HJp	St	Frac	{0, 1, ϵ}	N	Y	race conditions
Siminiceanu et al. (2012) Cataño et al. (2014)	Seq& Con	Plural	Pulse	St RGA	Sym	(U,I,S,F,P)	N	Y	deadlocks and race conditions correctness of the Plural specifications identifies Null-Pointer references.
Blom et al. (2014)	Con	OpenCL	Vercors	D	Frac	Perm(x,π) $\pi \in$ {rd, rw}	N	Y	race conditions for heap locations and functional correctness of GPGPU program.

Continued on next page

Table 6.2 – continued from previous page

Reference	Prog	Language	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Amighi et al. (2014)	Con	Java	Vercors	D	Frac	Perm(x, π) $\pi \in (0, 1] \cap \mathbb{R}$	N	Y	race conditions for heap locations and functional correctness of Java program.
Huisman and Mostowski (2015)	Con	Java (PSM)	KeY and PVS	D	-	readPerm(x,$Perm$) writePerm(x,$Perm$)$^{\alpha}$	N	Y	race conditions for heap locations. avoids reasoning overhead associated with concrete fractions
Amighi et al. (2015)	Con	Java	Vercors	St	Frac Count	$(0, 1] \cap \mathbb{R}$	N	Y	race conditions for heap locations.
Müller et al. (2017)	Seq	Silver	Viper	St	Frac	acc(x) acc(x, rd)	N	Y	race conditions for heap locations.
Ahmed and Cataño (2018)	Seq& Con	JML	Pulse	St	Sym	(U,I,S,F,P)	N	Y	correctness of JML specifications and verifying race conditions.
Sadiq et al. (2016)	Seq	Java	-	St	Sym	-	(U,I,S,F,P)	N	avoids annotation overhead by inferring symbolic permissions

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, Sym = symbolic, Fract = fractional, U = unique, I = immutable, S = share, F = full, P = pure, $\mathbb{Z}$ = set of Integers, $\mathbb{R}$ set of Real numbers, $\mathbb{N}$ set of positive Integers, NSL = new specification language, PSM = permission sharing model, PSAM = permission sharing and accounting model, x heap location, rd read access, C = session type contract, α a permission slice. $*$ permissions are read and write accesses, † implemented in the Fluid project, ‡ shows accessibility predicates.

location), if it has permission to do so. The approach defines three types of permissions based on their percentages: 'Full', 'Some' and 'No'.

The technique was realized in Chalice (Leino et al., 2009), a concurrent program verifier that supports programs with fork/join parallelism, monitors invariants and automatically verifies the absence of deadlocks and data races. In Chalice, programmers annotate programs with permission-based contracts using access predicates for each heap location. The annotation `acc(o.f)` represents 'Full' permission (100%) on a field of object `o` that shows that a thread has exclusive access on `o.f`. A fractional permission having `n` percentage of the actual permission is represented as `acc(o.f, n)`. A non-zero ('Some') permission depicts read-only access to location `o.f`, denoted as `rd(o.f)`.

Listing 6.7 shows a sample method specifications in Chalice. The pre-condition of the method `Clone()` in Line 4 specifies that the caller of the method must possess non-zero (read) permission on location `this.val` before calling this method. Following the Design by Contract principle, the post-condition in Line 5 specifies that the callee should generate Full permission on `result.val` field and return the input (read) permission to location `this.val`. Otherwise, the system will not be able to recover Full permission on it and in turn the location would remain immutable forever.

Listing 6.7: A sample program with accessibility predicates in Chalice (Leino and Müller, 2009).

```
1 class Cell {
2 int val ;
3 Cell Clone()
4 requires rd(this.val);
5 ensures acc(result.val) ∧ rd(this.val);{
6  Cell tmp := new Cell ;
7  tmp.val := this.val ;
8  return tmp ;}
9 }
```

The annotated program is analyzed to verify whether the code respects the permission contract for every thread schedule, as permissions flow between threads and monitors or between multiple threads. The analysis verifies that the sum of permissions for all threads remains less than or equal to 100% to ensure thread non-interference.

VeriFast

Jacobs et al. (2010, 2011) developed a sound, modular automatic program verification tool VeriFast to verify single- and multi-threaded programs written in C and Java. To enable verification, the programmer defines lemma functions in the program. Lemma functions are like ordinary C functions, except that lemma functions and calls of lemma functions are written within annotations. In VeriFast, lemma functions are interactively specified in the program following the Separation Logic style of specifications. They serve as proofs to ensure that a method terminates without producing any side effects in the system.

The approach simulates shared variables as heap locations and associates a permission coefficient using Boyland's fractional permission to each heap location to represent its access rights. The coefficient lies within (0,1] where 1 represents exclusive rights to manipulate a par-

ticular heap location and any value smaller than 1 represents a shared (read) access by multiple threads. The analysis works in a way that each method is symbolically executed based on other methods' contracts to verify its calls. The logic-based specifications are tracked through the system to detect exceptions such as `NullPointer` and `ArrayIndexOutOfBoundsExceptions` exceptions in the program, and to verify the domain specific problems in a program such as race conditions. The system does not support permissions for the program's local variables.

Heap-Hop

Villard et al. (2010) developed a program prover Heap-Hop* based on Hoare's monitors and copyless message-passing mechanism, an alternative to lock-based parallelism where only pointers to a message content in memory are transferred. The objective was to verify deadlocks, data races and to ensure the absence of memory leaks in heap manipulating concurrent programs, particularly those that involve communication protocols with list and tree structures. The approach uses *channels* as synchronization mechanisms where each channel consists of two endpoints, say e and f, dynamically allocated on the heap.

Heap-Hop requires programmers to specify pre- and post-conditions and loop invariants using separation logic (Reynolds, 2002). The communications between endpoints are governed using a contract C, a form of session types (Takeuchi et al., 1994), which specifies a valid sequence of message m passing on a channel. In Heap-Hop, ownership of cell to a heap location is represented using the notation $x\mapsto$ and the point-to relation $e\mapsto C\{a\}$ specifies a contract C in the state a with respect to a particular endpoint e.

Listing 6.8: A sample method with permission-based contracts in Heap-Hop

```
1 contract C {//session type contract
2 initial state a { !m ↦ b; !m ↦ c; }
3 state b {}
4 final state c {}
5 }
6 foo() { (e,f) = open(C); send(m,e); receive(m,f); close(e,f); }
```

The approach generates verification conditions based on the input specifications. It performs symbolic (forward) execution analysis of the generated conditions to determine what input (conditions) will cause each part of a program to execute and verifies the intended behavior of a program. The approach ensures that message sending never fails, and message reception should be blocked until the right message is received.

HIP/SLEEK

Hobor and Gherghina (2012) developed a Hoare-style concurrent separation logic that verifies Pthreads-style synchronization mechanism called barriers. Pthreads (POSIX Threads) is an API for threaded programming that manages various procedure calls for thread creation, destruction and synchronization (Butenhof, 1997). A common use of barrier calls is to manage a pool of threads in a pipeline. In Pthreads, barriers are used to redistribute ownership (as read and write access) of resources (memory cells) simultaneously between multiple threads.

*http://www.lsv.fr/Software/heap-hop/

At barrier calls, every thread gives up its write access to the portion of memory allocated to it, and gets back the read-only access to the entire memory.

The approach extends the concept of permission shares in DSA (Dockins at al., 2009) and assigns positive share to each thread to access a particular location. A full share is required to modify a particular location. A full share can be split into multiple partial shares that are merged back to get back the full share. The idea is to ensure that if a thread has a partial share for a particular location, no other thread has full share (permission) for that location.

Unlike previous Concurrent Separation Logics (OHearn, 2007; Hobor et al., 2008) that focuses on programs with critical sections, locks, and channels respectively, the approach uses barriers to model resource redistribution, and verifies if barriers are accessed safely in a concurrent environment. The idea is to associate some positive (fractional) share of the barrier itself as a pre-condition and to ensure that the sum of all preconditions entails full share of the barrier.

For example, the assertion barrier(bn, π, cs) defines a pre-condition that specifies a barrier bn with a positive share π having a state cs that holds before entering a barrier. The state of barrier changes as threads are released from the barrier, and the next stage will follow based on the post-condition barrier(bn, π, ns), when the state transitions to a new state ns. A `full` permission ensures that no thread has a 'stale' view of the barrier state to ensure thread non-interference. The approach extends HIP/SLEEK tool set (Gherghina et al., 2011; Nguyen and Chin, 2008) to verify concurrent programs with barrier calls. SLEEK is based on separation logic and HIP applies Hoare's rules to program verification.

Further, Jacobs and Piessens (2011) in their work on fine-grained concurrency, verified some of the program examples from HIP/SLEEK project. The objective was to discover a general mechanism that can implement barriers as locks, to reason about their correctness using the VeriFast tool. However, compared to HIP/SLEEK tool, Verifast poses annotation overhead. It is reported that "for 30-line example program, more than 600 lines of annotation are required in VeriFast" as user input to verify program behavior.

Pulse

Pulse (Siminiceanu et al., 2012) is an automatic formal verification approach and a tool that verifies the correctness of Plural (permission-based typestate contracts) specification itself, rather than the program implementation and its behavior. The goal was to write semantically correct specifications in order to verify program behavior based on these specifications. It supports five kinds of symbolic permissions: unique, full, share, pure and immutable in method specifications.

In Pulse, programmers specify design intents as permission-based typestate invariants and lock-based specifications at the code level to avoid deadlocks and data races. State invariants are used to enforce the properties that should hold true during program execution and to handle the design level inconsistencies in a program such as *Null* pointer references. Cataño et al. (2014) evaluated the efficacy and expressiveness of Plural specifications on a multi-threaded application called Multi-threaded Task Server (MTTS), to evaluate its design and verify its behavior using Pulse.

The Listing 6.9 shows lock-based typestate contracts in MTTS using Plural specifications.

Listing 6.9: Lock-based typestate contracts using access permissions in MTTS (Cataño et al., 2014).

```
1 @Perm(requires="Full(this) in NotAcq", ensures="Full(this) in Acq")
2 public abstract void acquire( ) { }
3 @Perm( requires="Full(this) in Acq", ensures="Full(this) in NotAcq"
4 public abstract void release( ) { }
5 }
```

Pulse defines lock-based specifications to ensure mutual exclusion to a critical section. The annotations 'Acq' and 'NotAcq' are used to represent the state of a lock i.e., to be acquired or not-acquired respectively. The locks are acquired using method `acquire()` in Line 2 and released using method `release()` in Line 3. The permission contract in Line 1, dictates that the `acquire()` method needs `Full` permissions, as pre-permission on the mutex (lock) object while acquiring a lock, and transitions it from 'NotAcq' into 'Acq' typestate. Similarly, the specification in Line 3 shows that the `release()` method, before releasing the lock, needs `Full` permissions on the lock object that is in 'Acq' state and transitions it from 'Acq' into 'NotAcq' typestate.

The code of the critical section is then enclosed between a call to method `acquire()` and a call to method `release()` to ensure mutual exclusion. The typestate transition in the given specification ensures that non-nested calls to method `acquire()` will always happen after a call to `release()` method. The permission contract ensures that if a thread has acquired a lock, it needs to be released before being used by other threads. However, as discussed previously, Plural does not support the reachability analysis of input specifications and cannot verify the absence of deadlocks caused by the input specifications. Pulse avoids deadlocks by using `try-catch-finally` statement in the code and enclosing call to method `release()` in a `finally` block to ensure that method `release()` is always called regardless of the termination status of the method.

Additionally detects violations of intended semantics using the model checking power of evmdd-smc (Roux and Siminiceanu, 2010) symbolic model checker. It helps programmers write semantically correct specifications and find possible concurrency at the method level, but to exploit the full potential of the Pulse tool, the programmers need to manually add permission-based typestate specifications in the source program, resulting in annotation overhead for the programmers. Moreover, the use of model checker can create state-space explosion problems even for a program of average size.

Ahmed and Cataño (2018) proposed an automatic translation technique that encodes JML-encoded Finite State Machine (FSM) specification of a Java program into Plural specifications (permission-based typestate contracts). The encoded specification was fed into Pulse to find problems such as unreachable states, unreachable methods and sink states (deadlocks) in the input specifications, and to reason about the correctness of the underlying program before it is implemented.

HJp

Westbrook et al. (2012) proposed a permission-based type system that supports task parallelism, array parallelism, and object isolation. The system called 'Habanero Java with permissions (HJp)' is an extension of their previous work on the Habanero Java (HJ) language. "HJ itself is a task-parallel extension of Java language" (Cavé et al., 2011). The system provides a practical solution to prevent data races for non-trivial parallel programs implementing multiple synchronization primitives, and parallel patters instead of just one.

The type system extends the Boyland's fractional permissions with two new permission types: *aliased write* and *storable* permission. Unlike previous approaches (Bierhoff and Aldrich, 2007) where write (unique) permission is only supported for non-aliased objects at any one time, the *aliased write* permission supports write operations on aliased objects. In the system, multiple threads can write on multiple objects without actually having unique permissions on them, as long as the permissions are not passed to other threads. The *storable* permission provides a new and simple way for expressing transitive permission in complex objects such as linked list. Storable permission associates permission to a 'whole tree of objects' instead of associating it to a single object This feature makes the technique different from existing approaches that require more technical machinery, and sound approximations, to manage permission transitivity in complex objects.

VerCors

Blom et al. (2014) proposed a simplified version of Kernel Programming Language (Betts et al., 2012) and a permission-based Separation Logic to reason about the correctness of the GPU kernel written in OpenCl[†]. As the GPU kernel extensively uses threads to support parallelism, the objective is to verify the functional correctness of GPU programs and to ensure data race freedom in the underlying architecture.

The work follows an earlier work of Haack and Hurlin (2008) on verification of the muti-threaded programs in which a thread can only access or update a particular memory location if it has permission to read or write. Multiple threads with read permissions can access the same location but only one thread can hold write permission at a time to change its content. The same idea was applied to delegate permissions across work groups and then to distribute permissions over threads. The approach was validated using VerCors[‡] (Amighi et al., 2012), based on permission-based Concurrent Separation Logic (Haack et al., 2008), and uses Silicon (Juhasz et al., 2014) as a back-end verification tool which natively supports an expressive permission model.

The VerCors verification approach (Amighi et al., 2014) supports multiple synchronization primitives and exploits the verification capability of JML (Leavens et al., 2006), to reason about the functional correctness of the underlying program. It extends JML specifications with fractional permissions and uses the conjunction operator $*$ in separation logic to define permission-based contracts as JML comments in a Java program. Permissions on a particular field are spec-

[†]Khronos OpenCL Working Group, The OpenCL specification, http://www.khronos.org/opencl/
[‡]https://fmttools.ewi.utwente.nl/redmine//projects/vercors-verifier/wiki/Puptol/

ified using propositional formula `Perm(e.f, π)` where π represents fractional permission in the range $(0, 1]$ assigned to the particular field f of object e. This permission is then transferred between threads at synchronization points and analyzed with the execution of the program.

Listing 6.10 shows a sample Java class `point` in 2D. A state predicate `state(frac p)` in Line 2 specifies that `p` permission is required on disjoint locations, `this.x` and `this.y` in memory. These predicates are then used to specify permission contract for the same locations at method level. For example, the pre-condition `state(1)` of method `set()` in Line 5 specifies that the method requires full (write) permission on locations `x` and `y`, and the post-condition `ensures state(1)` ensures that the method returns the same permission on the corresponding locations when it exits. The invariant clause, in Line 4 specifies a functional property that both points should be in the first or third quarter of its cartesian space.

Listing 6.10: A Java class Point example in (Amighi et al., 2014).

```
1 public class Point{
2 //@ resource state(frac p) = Perm(this.x, p) * Perm(this.y, p);
3   private int x, y;
4 //@  invariant (x >= 0 && y >= 0)  (x <= 0 && y <= 0);
5 //@  requires state(1); ensures state(1);
6  public void set(int xv, int yv){ this.x = xv; this.y = yv; }
7 //@ given frac p; requires state(p); ensures state(p);
8 public void plot(){}
9 //@ given frac p; requires state(p); ensures state(p);
10 public int getQuarter(){}
11 }
```

The contracts of methods `plot()` and `getQuarter()` in Line 8 and Line 10 respectively, specify that both methods require read permission `p` on locations `x` and `y`, which means that they can be executed simultaneously by multiple threads without the fear of data races. This is because the the pre-conditions of both are disjoint with respect to memory. This is not true for the method `set()` as it requires full permission on the same locations.

Instead of simply defining the amount in fractions of permission transferred, Huisman and Mostowski (2015) extended the previous fractional permission model in VerCors by having symbolic expressions which include the kind of *transfer* applied to permission, and the owner of the transferred permission. The approach facilitates high-precision, complex synchronization scenarios in concurrent data structures, and supports permission tracking at a high level of abstraction as compared to the previously mentioned approaches such as Veri-Fast (Jacobs et al., 2011) and Chalice (Leino et al., 2009).

In the approach, the program is annotated with symbolic (permission) expressions using JML annotations in the functional style. The analysis then tracks the owners using permission expressions and checks their permission return paths to reason about their behavior. The system identifies permission owners using object references and manages a list of owners. Whenever permissions are assigned to some owner, it is being added in the list and when an owner returns permissions, it is removed from the list. Each owner is considered as a permission slice. If all slices refer to the same owner, it means that the owner would have full permission. Otherwise, access is partial (read).

The proposed permission theory was formalized in the KeY tool (Beckert et al., 2007), an interactive verifier for Java that is based on dynamic logic. The system extends KeY tool

with permission accounting to verify program properties that are based on purely first-order reasoning. The general properties that require structural induction proofs are validated using the PVS tool (Owre et al., 1992), because of its automated deduction and theorem proving capability.

Amighi et al. (2015) proposed a variant of OHearn's Concurrent Separation Logic (OHearn, 2007) to perform practical reasoning of Java-like concurrent programs having main concurrency primitives such as dynamic thread creation, thread joining, wait-notify scenarios and lock reentrant mechanism. It combines Parkinson's share model with Boyland's fractional permissions to avoid data races.

Like Parkinson's share model, access for a particular heap location is maintained using a resource's invariant property, where 1 represents full (exclusive) permission to a heap location, and a fractional value in the interval (0, 1) defines the concurrent read access of a particular location. The idea is that a thread having partial permission is not allowed to write on a heap location and the total permission to access a heap location cannot exceed 1. Like the OHearn's approach, when a thread acquires a lock, it gets access to part of a heap location specified as a resource's invariant property. Upon unlocking, it transfers access of the same resource back to lock, to re-establish the resource's invariant property. The permissions are transferred between threads at the time of thread creation, thread joining and at lock entrances and reentrance points.

Viper

Müller et al. (2017) developed a verification infrastructure called Viper that encodes permission reasoning in an intermediate language. The infrastructure includes two back-end verifiers and four front-end tools for Chalice, Java, Scala, and OpenCL that was developed as a part of VerCors project (Blom and Huisman, 2014). It targets a sequential, object-based intermediate language also called Viper. A Viper program does not have classes and an object can access every field declared in a program. Moreover, there is no implicit receiver object for methods and functions.

In a Viper program, a programmer defines accessibility predicate (Cousot and Cousot, 1977) to specify permission-based pre- and post-conditions and loop invariants for heap structures. A method can access a particular heap location if the appropriate permissions are held by that location and permissions are then transferred between method execution and the loop body to verify program behavior based on the specifications.

Listing 6.11 shows a sample sorted integer list `data` with access predicates in a Viper program, to verify its functional behavior. The macro `sorted(s)` in Line 2 sorts input list `s` in ascending order. The `insert` method adds a new element `elem` in the `Ref` list and returns the index `idx` where the new element was inserted.

Listing 6.11: A sorted integer list and its specifications in Viper (Müller et al., 2017).

```
1 field data: Seq[Int]
2 define sorted(s) forall i: Int, j: Int :: 0 <= i && i < j && j < s
3                                         ==> s[i] <= s[j]
4  method insert(this: Ref, elem: Int) returns (idx: Int)
5  requires acc(this.data) && sorted(this.data)
```

```
6  ensures acc(this.data) && sorted(this.data)
7  ensures 0 <= idx && idx <= old(this.data)
8  ensures this.data == old(this.data)[0..idx] ++
9              Seq(elem) ++ old(this.data)[idx..]{
10  idx := 0
11  while(idx < this.data && this.data[idx] < elem)
12  invariant acc(this.data, 1/2)
13  ...
14  { idx := idx + 1 }
15  ...
16 }
```

The pre-condition of method `insert()` in Line 5 specifies that the method requires full permissions on the object list `this.data` and it should be sorted. The post-condition in Line 6 guarantees that when the method exits it returns the sorted list to the caller with the consumed permission. The second post-condition in Line 7 constrains and thus validates the index, while the third post-condition in Line 8 relates the current state of the list with the method's pre-state, using an `old` expression.

The `insert()` method iterates over `data` list to determine where to insert the new element `elem` in Line 11. The loop invariant (Line 12) specifies that loop body needs a half (read) permission on the list, while the second half permission would be held by method execution to ensure that the loop body does not modify the list. The Viper's front-end tools then encode the annotated program into an intermediate language acceptable by the by the back-ends tools, to verify its behavior.

6.4 Automatic Inference of Access Permissions

Permission-based access notations have been generated as means for program verification in many approaches (Bierhoff et al., 2009a; Leino et al., 2009; Ferrara and Müller, 2012; Le et al., 2012; Heule et al., 2011, 2013; Sadiq et al., 2016; Dohrau et al., 2018). The generated specifications are either in the form of read/write accesses, fractional or symbolic permissions. The overall goal of these approaches was to relieve programmers from specification overhead resulting from manually adding permission-based annotations in a source program for verification purpose. Table 6.3 shows a summary of the work done to infer permission-based specification in sequential and concurrent programs.

Table 6.3: Access permission inference for sequential and concurrent programs.

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Bierhoff et al. (2009a)	Seq	Plural (NSL)	Plural	(St,D)	Sym	(U,I,S,F,P)	Frac	Y	protocol verification
Leino et al. (2009)	Con	NSL	Chalice	D	Frac	acc(x) rd(x)‡	access pure	Y	race conditions & deadlocks
Heule et al. (2011) Heule et al. (2013)	Con	-	Chalice	St	Frac	acc(x, 1) acc(x, rd)	full read	Y	race conditions
Le et al. (2012)	Con	NSL	VPerm	D	Frac	@full[ν] @value[ν]$^{\phi}$	full zero	Y	race conditions.
Ferrara and Müller (2012)	Con	Scala	Sample	St	-	acc(x, p) p $\in$ (0, 1] $\cap\mathbb{R}$ p $\in$ (0,1] $\cap\mathbb{Z}$, p $\in$ (0, 100]%	Frac Count Chalice	Y	race conditions.

Continued on next page

Table 6.3 – continued from previous page

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno	Properties
Dohrau et al. (2018)	Con	Viper	Scala	St	Frac	[0,1]∩ℝ	read and write	Y	race conditions
Sadiq et al. (2016)	Seq	Java	Sip4J§	St	Sym	-	(U,I,S,F,P)	N	-

Keys to the table: Seq = sequential, Con = concurrent, St = static, D = dynamic, *Sym* = symbolic, Fract = fractional, U = unique, I = immutable, S = share, F = full, P = pure, ℤ = set of Integers, ℝ set of Real numbers, NSL new specification language, x heap location, ν represents a non-heap location. *rd* for the read access. ‡ accessibility predicates.

6.4.1 Inference of Read & Write Accesses

Chalice (Leino et al., 2009), a concurrent programming language uses autoMagic, a command-line option, to infer the read and write accesses for the heap locations specified with accessibility predicates as previously explained in Section 6.2. The inferred notations are in the form of `pure` and `access` notations that represent `read-only` and `full` permission respectively for the specified heap locations.

Le et al. (2012) proposed a new permission system to avoid data races in multi-threaded applications having fork/join parallelism. The objective was to ensure the absence of data races for program variables that are not actually heap variables but can be accessed by multiple threads.

The scheme infers variable permissions at the method level using `procedure specifications`. However, in the procedure specification, a programmer explicitly specifies state changes (if any) for the referenced variable accessed by the current thread, using permission-based state invariants without actual variable permission. The generated permissions are in the form of notations such as `full` or `zero` where `full` represents exclusive rights on the referenced variable and `zero` represents the absence of permission. The proposed technique then tracks permission flow between threads to ensure safe access to the shared variables.

Listing 6.12 shows an example procedure specification example in a sample fork-join program. The method `creator()` takes two variables `x` and `y` as reference parameters. The `requires` clause in Line 2 specifies that the method needs `full` permission on the referenced variables `x` and `y` as pre-permissions when the method is called. The `ensures` clause specifies that the method should generate the same permissions as post-permissions on the same referenced variables when it exits (Line 3). The state changes are represented using prime `′` notation. For example, the specification "$y' = y + 2$" in Line 3 specifies state changes for the referenced variable `y` that should hold after the method completes its execution. These specifications are then tracked in the system to generate actual variable permissions for the referenced variables.

Listing 6.12: A fork/join program fragment with procedure specifications (Le et al., 2012).

```
1 int creator(ref int x, ref int y)
2 requires @full [x, y]
3 ensures  @full [y] ∧ y′ = y + 2 ∧ res = tid and @full[x] ∧ x′ = x + 1 ∧ thread =
      tid;{
```

§https://github.com/Sip4J/Sip4J

```
4  int tid = fork(inc, x, 1);
5  inc(y, 2);
6  return tid;
7 }
```

The proposed scheme was realized in a concurrent program verifier called Vperm[¶] that verifies the correctness of concurrent applications written in C/C++ language. The approach does not handle phased access to a shared variable by multiple threads, in which case a translation algorithm is used to simulate the affected variables as pseudo-heap locations.

6.4.2 Inference of Fractional Permissions

Bierhoff et al. (2009a) proposed a deterministic algorithm to infer permission flow through the program while verifying usage protocols. The objective was to avoid the permission tracking overhead associated with splitting and joining the fractional permission during verification.

The algorithm is implemented in the Plural tool (Bierhoff and Aldrich, 2008) that performs dataflow analysis of the program with `in` and `out` permissions as developer-provided annotations. The system collects linear constraints over fractional variables by tracking the flow of permissions in the program. The analysis then ensures the satisfiability of constraints in a modular fashion. The approach supports polymorphism over fractions that not only facilitates modular reasoning of the program, but also avoids imprecision in loops by allowing permission consumption inside loops. Furthermore, the technique automatically infers loop invariants in a program.

Ferrara and Müller (2012) proposed a permission inference technique to infer fractional and counting permissions for heap locations in a class-based language having threads and monitors. The technique performs static analysis of the source program and inference is based on abstract interpretations (Cousot and Cousot, 1977), a theory for defining and soundly approximating the semantics of a program. The approach firstly computes symbolic values (approximations) for each heap location using the pre- and post-conditions and lock invariants defined at the method and class level respectively. It then infers constraints over these symbolic values to reflect permission-based intermediate representation for the heap locations. Finally, it generates specifications in the form of fractional (value between 0 and 1) and counting (value between 0 and Integer:MAX_VALUE) permissions for each heap location in the program.

The symbolic permissions ($\overline{AV}$) for each heap location are calculated as "the summation of symbolic values s_i multiplied by integer coefficients a_i (to represent how many times the permission is consumed or returned) and an integer constant c" (see Formula 6.4). The integer constant `c` represents `full` permission that is inhaled when an object is created.

$$\overline{AV} = \sum_i a_i * s_i + c, \text{where } a_i, c \in \mathbb{R}, s_i \in SV \tag{6.4}$$

[¶] http://loris-7.ddns.comp.nus.edu.sg/project/vperm/.

For example, the expression $1 * \texttt{Pre(C,m,c:f)} + 1 * \texttt{MI(C,c:f)} + 0$ represents symbolic permissions computed for each heap location (`c:f`) in method `m()` of class `C` where `Pre(C,m,c:f)` represents the symbolic value (s_i) assigned to location (`c:f`), as pre-condition before acquiring a lock, and the notation `MI(C;c:f)` represents monitor (`MI`) acquired on location `c:f`. Further, the notation `Post(C,m,c:f)` represents a symbolic value assigned to a heap location,as post-condition, to get the original permission back on it when the monitor is released. The technique then infers constraints over these symbolic values to generate actual permissions as fractional permissions.

The inference technique is implemented in `Sample` (Static Analyzer of Multiple Programming LanguagEs)[‖] that supports programs written in Scala (Odersky et al., 2004). Listing 6.13 shows the OwickiGries (Owicki and Gries, 1976) program fragment as an input program. In the example, all expressions are self explanatory except the old expression `old (c.c1)`, in Line 4, that allows post-conditions to refer back to the pre-state of a referenced variable and its associated predicates.

Listing 6.13: The OwickiGries program (fragment) with method level specifications in (Ferrara and Müller, 2012)

```
1 class W1 {
2 var c : Cell;
3 method Inc()
4  ensures c.c1 == old(c.c1) + 1{
5  acquire c;
6  c.c1 := c.c1 + 1;
7  release c;
8  }
9 }
```

In method `Inc()`, between `acquire c` and `release c` clauses (Line 5 and 7), the current thread is assigned with a symbolic permission $1 * Pre(W1,Inc,c:c1) + 1 * MI(Cell,c:c1)$ for location $c:c1$. When method exits (Line 9), is gets the symbolic permission $1 * Pre(W1,Inc,c:c1) = 1 * Post(W1,Inc,c:c1)$ as post-condition since the monitor of `c` is released. Solving constraints over these symbolic values, the system generates `full` (1) as fractional permission for location `c:c1` when method completes its execution and control is passed to Line 8.

The system works very well for Chalice lattice domain. However, teh analysis based on fractional specifications is challenging and sometimes, the system converges the generated permissions back to zero to explicitly terminate analysis. Moreover, it provides limited support to infer permission contracts for programs having recursive data structures. The rate of inferring permission contracts for such programs is at minimum 36% and 68% at maximum.

In an extended work of Ferrara and Müller's permission inference, Dohrau et al. (2018) proposed a static analysis to infer permission-based contracts for array manipulating concurrent programs. The technique is based on Separation and related logics (Reynolds, 2002; Smans et al., 2009). The idea is to explicitly associate a separate (fractional) permission for each array element to specify its accessibility by parts of the program. The value 1 represents `full` access while `rd`, a positive fraction of permission, represents the concurrent (read) access to a memory location. The analysis then infers read and write accesses for the specified

[‖] http://www.pm.inf.ethz.ch/research/sample.html

memory locations to generate permission contracts at the method-level and within loop.

For example, the approach associates each array element say $q_a[q_i]$ with a fraction of permission using a conditional expression of the form $q_a = array \wedge q_i = index \ ? \ 1 : 0$ that specifies `full` permission (1) for element $array[index]$ and no permission for all other elements. The permission required for each loop iteration is computed using a `maximum expression` that calculates the maximum of permission required by each referenced variable changed in a particular loop iteration. The whole (complete) loop execution depends on the maximum of all the fractions over all loop iterations. It is used to infer read and write specifications for all indices of an array, for the whole loop.

However, it is generally acknowledged that tracking concrete fractional values is a cumbersome task for programmers especially when fractions continue to decrease indefinitely for a particular scenario (Heule et al., 2013). Moreover, the use of fractional permissions makes the specifications too low-level which can be tedious to add manually and harder to reuse and adapt for programmers.

6.4.3 Inference of Symbolic Permissions

Heule et al. (2011, 2013) proposed a technique to automatically convert fractional permissions into abstract read/write permissions for shared-memory concurrent programs. The objective was to specify concurrent constructs such as fork/join threads, locks/monitors with abstract permissions to avoid the complex reasoning overhead associated with fractional shares.

The abstract read permissions allow programmers to reason at a high-level of abstraction than using the fractional values for reading. The objective was to avoid the complex reasoning overhead associated with handling concrete values in fractional permissions during verification. The proposed methodology is implemented in Chalice. The system generates two kinds of permissions i.e., `full` and `read`. Like Chalice, it takes a program annotated with accessibility predicates such as `acc(x.f,1)` and `acc(x.f, rd)` at method level. The value 1 is mapped to represent the `full` (read and write) permission and `rd` represents the shared `read` permission (a part of permission that is not full) for the referenced object `x.f`. Moreover, the system automatically computes read (`rd`) permission instead of programmers having to compute this value explicitly.

6.5 Access Permission-based Parallelization of Sequential Programs

Pure functional programming is a paradigm where languages are suitable to develop concurrent applications as they do not allow side effects. This means that two parts of the code (methods) cannot access the same mutable states and therefore do not interfere with each other. To avoid side effects because of shared states, pure functional languages create copies of objects whenever there is a change in an object's state, thus creating immutable objects. Therefore, for these languages, the compiler freely parallelizes execution of the program without causing side effects (Hughes, 1989). Although pure functional languages make concurrent programming

more deterministic, cleaner and safer, but for complex mutable objects (reference types), these languages have historically low performance and results in excessive memory consumption.

In the real world where most of the realistic applications have already been written in non-pure (traditional) imperative programming paradigms, the challenge is to exploit the implicit concurrent behavior of the underlying program, having mutable states and complex objects. With this consideration in mind, research projects such as `Concurrency Made Easy` [**] and `UPScale` [††] are still investigating ways to make concurrent programming more straight forward and more reliable. In the past few years, some permission-based programming languages, compilers, and runtime systems have been developed (Jones, 2003; Aldrich et al., 2011, 2012; Stork et al., 2009, 2014; Rafael et al., 2014; Fonseca et al., 2016).

Table 6.4 shows a summary of the related work studies in this research that automatically parallelize the execution of sequential programs based on access permission rights.

Table 6.4: Access permission-based program parallelization.

Reference	Prog	Lang	Tool	Analy	Perm-Kind	Perm-Specs	Perm-Infer	Anno
Jones (2003)	Seq	Haskell	-	st	read write	monads	N	Y
Aldrich et al. (2011) Aldrich et al. (2012)	Seq	Plaid NSL	Plaid	st	Sym	(U,I,S)	N	Y
Stork et al. (2009) Stork et al. (2014)	Seq	Æminium (NSL)	java2 aeminium	st	Sym	(U,I,S)	N	Y
Rafael et al. (2014) Fonseca et al. (2016)	Seq	Java	J2Par	st	-	read write	N	Y

Keys to the table: Seq = sequential, St = static, *Sym* = symbolic, U = unique, I = immutable, S = share, F = full, P = pure, $\mathbb{Z}$ = set of Integers, $\mathbb{R}$ set of Real numbers, NSL = new specification

Haskell is a pure functional programming language that provides constructs to deal with mutable states and side-effects (Jones, 2003). In Haskell, all side effects are explicitly mentioned in the method's signature. While accessing a shared resource, Haskell uses the concept of access permissions to assign access privileges for functions. For instance, if a function wants to perform an I/O operation, it requires an I/O monad which is a global permission that specifies access to all the global states of a system. In Haskell, I/O monads are used to avoid race-conditions and parallelize the execution of functions but the system has only one type of permission for the entire system, which may create performance bottleneck for highly complex and concurrent applications.

Aldrich et al. (2011) developed a new permission-based programming language, a type system and a runtime system called Plaid, that leverages the flow of permissions through the system, to support automatic parallelization of typestate-based sequential programs. The aim was to extend the typestate-oriented programming (Aldrich et al., 2009) with first class-states and access permissions. Plaid was originally designed to track the typestate of a reference object at runtime and to handle unrestricted aliasing using permission-based typestate information.

However, in Plaid, programmers explicitly add permissions-based typestate information

[**] Concurrency Made Easy, http://cme.ethz.ch/.
[††] UpScale, https://upscale.project.cwi.nl/

(pre- and post-state) to show how a method changes the state of a receiver object, using state transition operator ('≫'). The syntax and semantics of the Plaid language are already explained in Section 2.3.

Figure 6.14 shows a sample code fragment for the buffer implementation, with permission-based typestate annotations, in Plaid. A buffer can be in one of the two states i.e. `EmptyBuffer` and `FullBuffer` (Line 1). The method `put` is associated with the empty buffer. The signature of the `put()` method (Line 4) specifies that state of receiver object should change from `EmptyBuffer` to `FullBuffer` when buffer receives some element. The state `FullBuffer` requires a field element `elem` that is passed as a method parameter `e`. The permission-based contract (Line 4) specifies that the passed element has `unique` permission in `Element` state, and the method does not return any permission (say `none`) to the caller, because a field reference with exclusive rights (`unique`) was created for that element in the `FullBuffer` state. Otherwise returning permission back to the caller would cause a violation of the uniqueness property. Likewise, `FullBuffer` state has a single operation `get()` (not included in the example due to brevity), which returns the current value of the object represented by the reference variable `elem`, and ensures, that the receiver object goes back to the `EmptyBuffer` state.

Listing 6.14: A buffer implementation (fragment) in Plaid (Aldrich et al., 2011).

```
1 state Buffer comprises EmptyBuffer, FullBuffer {}
2 ...
3  state EmptyBuffer caseof Buffer {
4  method void put(unique Element ≫ none e) [EmptyBuffer ≫ FullBuffer] {
5    this ← FullBuffer {elem = e};
6   }
7 }
```

The Plaid type checker then leverages the access permission flow through the system along with the associated typestate information to ensure protocol compliance at runtime. The Plaid type system allows permission splitting, joining and type-casting automatically (when and where required) using access permission splitting and joining rules given in Table 2.1. Plaid follows the Design by Contract principle in general, to verify permission-based contracts at the method level. Further, Plaid's runtime dynamically verifies the typestate usage, and checks permission-based dependencies to parallelize execution of the underlying program accordingly.

The limitation of Plaid is the need to manually annotate program with permission-based typestate information, which creates annotation overhead for the programmers. Moreover, it is mentioned that "the current implementation of type checker does not support type checking at every program place". Hence, the type checker needs to be enhanced for better results. Further, the inclusion of other access permissions such as `full` and `pure` in the type system is highly desirable.

Incorporating a language runtime support for permissions in Plaid, Stork et al. (2009, 2014) developed a permission-based programming paradigm called Æminium to develop by-default concurrent applications. In Æminium, to achieve implicit concurrency, the idea is to to make implicit dependencies explicit based on access permissions, thereby avoiding side effects in the system and parallelizing the code to the extent permitted by permission-based dependencies. However, like Plaid, it requires programmers to explicitly specify permission-based annotations in the source program but unlike Plaid that uses access permissions to

track the states of the referenced objects, Æminium uses access permissions to avoid side effects in a program. A `unique` and `immutable` permission specifies the `'exclusive'` and `'read-only'` access on a referenced object respectively whereas a `share` permission specifies concurrent access for the shared objects. In Æminium every mutable (shared) object is assigned a data group (e.g., α). "A data group represents an abstract collection of objects" (Leino et al., 2002). Æminium defines three types of permissions for a data group collectively called data-group permissions i.e., `'exclusive'`, `'shared'` and `'protected'`. Access to a data group (α) is handled using `atomic` and `share` blocks as mutual exclusion primitives.

Æminium performs static analysis of a permission annotated program using the Plaid compiler. It tracks permission flow through the system (for methods' parameters and receiver objects) to compute data dependencies at the task-level. This information is then used to parallelize execution of a sequential program on top of Æminium runtime.

Æminium provides concrete design and semantics of the proposed by-default concurrent paradigm, however, the advantage of automatic parallelization without data races comes at the cost of high annotation overhead and a sophisticated type system. This annotation overhead becomes even more significant as the program size increases. Therefore, automatic inference of permissions-based annotations (the goal of this thesis) is highly desirable.

Inspired from permission-based automatic parallelization of sequential Java programs in Æminium, Rafael et al. (2014) and Fonseca et al. (2016) developed a technique and a tool called `JPar` to perform instruction-level automatic parallelization of sequential Java programs, based on the Æminium compiler (Stork et al., 2014), the Æminium runtime and ÆminiumGPU (Fonseca and Cabral, 2013). The aim was to improve fine-grained automatic parallelization of sequential programs and improve performance on multi-core architecture while consuming less memory.

The JPar compiler uses data group abstraction (Leino et al., 2002) to check side effects in the program. In JPar, data groups represent the memory sections shared between different parts of the code. The proposed approach performs static analysis of the source code based on the application's Abstract Syntax Tree (AST). It extracts the instruction's signatures which represent the data and control dependencies (read, write, control flow information, etc.) on data groups. Each Abstract Syntax Tree node is then annotated with the inferred dependencies. The permissions on data groups are depicted as `read(dg)`, `write(dg)` or `control(dg)`, where `dg` represents a particular data group.

Listing 6.15 shows a Fibonacci program with instruction-level signatures and with data and control dependencies, as data group permissions, in JPar.

Listing 6.15: A Fibonacci method with instruction-level data and control dependencies in JPar (Fonseca et al., 2016).

```
1 int f(int n) {
2  if (n < 2)// read(n), control(f)
3    return n; // write(return), control(f)
4  int a = f(n - 1); // call(f), read(n), write(a)
5  int b = f(n - 2); // call(f), read(n), write(b)
6  return a + b; // read(a), read(b), control(f), write(return)
7 }
```

The annotation `read(n)` in Line 2 shows that the AST node reads the memory location `n`. Likewise, the annotation `write(a)` in Line 4 depicts that the subtree writes on reference `a`. The annotation `control(f)` denotes the control flow of other operations in the method `f`. This is generally the case for `return`, `continue`, `break`, `switch`, `selection` and `iterative` statements (Line 3 & 6). The annotation `call(f)` represents a method call for method `f` (Line 4 & 5). To achieve task-level parallelism, the proposed approach generates task-based Java code from the sequential Java version, where each task respects the inferred read, write and control dependencies in the program. However, instead of inferring access permission-based dependencies from the source program, the approach infers the dependencies as read, write and control flow information for each task and task parallelism is achieved using the ÆminiumGPU runtime system, a new programming paradigm.

6.6 Research Challenges

The study of existing permission-based approaches shows that although access permissions have been used to provide a sound reasoning mechanism to verify program behavior and achieve concurrency, the existing permission-based approaches, in the literature, are still limited in their support due to the following reasons.

Permission Annotation Overhead. The common problem with all the permission-based verification approaches is the annotation overhead associated with the need to manually add permission-based dependencies (invariants, contracts, assertions, etc.) or other access notations, to explicitly specify state changes and grant or restrict access to multiple references (threads), on the shared memory locations. It is generally acknowledged that manually annotating programs is laborious, challenging and time-consuming.

Permission Verification Overhead. Given the intricacies in creating manual permission-based specifications, programmers are very likely to omit important dependencies or create misspelled specifications that may again lead to problems such as data races and deadlocks and may in turn pose verification overhead. There is no guarantee that the written specifications are correct. Although some existing approaches presented a solution to this problem, by identifying the misspelled (missing) specifications and by verifying that, the specifications follow the intended semantics, the techniques themselves are limited in ensuring whether the program implementation complies with the input specifications and vice versa. Furthermore, although, access permissions support modular reasoning of a program behavior without analyzing the entire program analysis, certain program properties such as global invariants and liveness, are hard to verify.

Permission Tracking Overhead. Existing verification and parallelization approaches use different forms of permission types such as fractional permissions, based on their expressiveness and to facilitate the ease of analysis. The runtime systems then analyzes the permission flow through the system to verify program behavior against the specification and computes the data dependencies in the system based on the specification.

In particular, fractional permissions use fractional style to express the access rights for

a reference in the range (0,1] but its analysis is challenging for programmers, due to the overhead associated with tracking the splitting and the joining of concrete values. It is generally acknowledged that tracking fractional values in a program is a cumbersome task for programmers. The situation becomes more serious when fractions are split into multiple levels, which may create concerns relating to the proper termination of the analysis and affects the soundness of the technique itself.

Counting permissions are complementary to fractional permissions but they do not support all types of synchronization primitives. Symbolic permissions combine the access rights and aliasing information of a reference and have been used to allow programmers to reason about the program correctness, against specification at a higher level of abstraction than fractional or counting permissions. Therefore, automatic inference of permission-based specifications in the form of symbolic values is desirable to free the programmers from the low-level analysis overhead associated with adding and tracking concrete values in the program.

In addition to the specification overheads and related problems discussed above, the following factors may also hinder the wider adoption of existing permission-based verification approaches.

Languages and Tools. Existing permission-based verification and parallelization approaches are mostly based on formal specification languages and type systems to support access permission as part of the language. It can be challenging for most programmers who may not be adept at the new syntax and semantics in order to exploit their functionalities. Furthermore, most of these approaches are either research prototypes or developed in languages that are not commonly used for general-purpose software development. These factors could limit the adoption of the existing approaches to verify programs written in mainstream programming languages such as Java.

Program Constructs. Existing verification approaches have limited support for synchronization constructs such as fork-join parallelism, atomic blocks or semaphores. There is also limited support for the recursive data structures. Most of the approaches support verification of heap locations, while investigation on non-heap locations has not been as prevalent. These limitations restrict their ability to verify real-world applications.

Program Analysis. Existing approaches either perform static and dynamic program analysis or employ model-checking techniques to verify program properties. All have their own pros and cons that affect the scalability of these approaches when analyzing program constructs and verifying the program behavior. For example, the techniques employing model checkers may face state-space explosion problems even for a program of average size.

Properties. Most verification tools focus on certain aspects of the program behavior, such as verifying the correctness of API protocols or avoiding synchronization issues such as data races are just two aspects. Some tools can address a combination of issues, but none of them cover all aspects of a program behavior.

The study shows that existing permission-based approaches individually cannot provide a complete solution to program verification or parallelization, without incurring additional cost to programmers or verifiers. Overall, given the number of different permission-based formal type theories and programming models that received remarkable attention over the last decade in the research community, there is an impending need to make these approaches adoptable and adaptable for general-purpose program development, and verification using mainstream programming languages such as Java, C++ and .Net.

The common problem in all the existing approaches is the annotation overhead associated with the need to manually add the permission-based specifications in the source program. Therefore, the automatic inference of permission-based specifications from the source program (the goal of this thesis) can be the first step to exploit the verification and parallelization power of these approaches, without posing any extra burden on programmers, to enhance their applicability in the IT industry.

These requirements are addressed by our permission inference framework `Sip4J`, by inferring access permissions without using any method level specifications as well as making the technique suitable for mainstream programming languages.

The next step can be an integration of the most widely used permission-based verification tools such as Plural, Pulse, Verifast and VerCors, that at least support a common programming models such as Java, and employ the concurrency-by-default approaches such as Plaid and Æminium, to parallelize execution of programs written in mainstream programming languages. The ideal solution to all the above challenges can be the integration of the commonly used abstractions such as typestates and access permissions, as first-class language constructs in the mainstream programming models, to develop a complete, sound, modular, automated and economically feasible framework for everyday program development and verification.

Chapter 7

Conclusion

It has been predicted and observed that Moore's Law will continue to hold for only another five to ten years. As a response, the focus of modern processor design has shifted from increasing clock speed of individual physical cores to increasing the number of cores, hence increasing the potential for parallel execution of application software. To exploit parallelism offered by multi-core processors, mainstream programming languages such as Java typically make use of explicit concurrent programming constructs such as threads and locks to explicitly specify the side effects and define the synchronization strategies. However, such constructs give rise to significant code complexity and synchronization errors. Therefore, access permissions (Aldrich et al., 2011; Stork et al., 2014), as an alternative mechanism, to achieve implicit concurrency present in single-threaded applications.

The study of the literature (Chapter 6) shows that access permissions are powerful mechanisms that represent and combine the read and write behavior of a referenced object as well as its aliasing information thereby the side effects. Access permission sharing and accounting models (Boyland, 2003; Bornat et al., 2005; Bierhoff and Aldrich, 2007) attained considerable attention in the research community in the last few decades, because of their rich expressiveness and sound reasoning capabilities, to specify and verify the correctness of shared-memory concurrent programs. Furthermore, as access permissions can perform method (read and write) operations in a non-interfering manner, they have been used to achieve the implicit concurrency from sequential programs without explicitly introducing the concurrency constructs in the program.

However, to exploit the benefits of access permissions, all the existing approaches pose programmers with the annotation overhead for manually adding permission-based dependency information in the program. These approaches not only requires programmers to be adept to completely new formal type theories and programming paradigms, but they also need to identify and add specifications in the source program which is a time consuming and error-prone task. Given the intricacies in creating these constructs, it is very likely for a programmer to omit important constraints or create misspelled specifications that may again lead to problems such as race-conditions or deadlocks due to the wrong dependencies. These issues have hindered the wider adoption of the existing approaches for the general-purpose program development and verification purpose. Ideally, it would be much easier and more effective for a programmer to not to be concerned with manually specifying permission-based annotations in the program, while still being able to exploit the benefits offered by such annotations.

7.1 Thesis Contributions and Reflection

This thesis aims to free programmers from the annotation overhead for manually adding permission-based dependencies in the source program The objective is to develop a fully automated permission inference and checking framework, to infer access permission contracts from the source code of single-threaded Java programs in a correct and efficient manner.

To the best of our knowledge, the work presented in this thesis is the first attempt to infer symbolic (access) permission contracts for programs written in a mainstream programming language. The core functionalities of the proposed approach have been realized in a prototype tool, `Sip4J`, along with its integration with a permission-based model checking tool Pulse. The empirical evaluations have shown the benefits of the proposed approach and the `Sip4J` framework itself for the permission-based research community and as a whole to the IT industry.

The main contributions of this thesis are as follows.

1. **A permission inference methodology** that reveals the access permission-based dependencies from the source code of a Java program and maps them in the form of access permission contracts (**RQ1**).

   The permission inference approach (Chapter 3) performs modular static analysis of an un-annotated Java program to automatically reveal the method's side effects and maps the object's accesses, in the form of five types of symbolic permissions (Section 2.1), at the field level. The generated specifications are then automatically mapped in the form of access permission contracts, as pre- and post-permissions, following the Plural specifications (Section 2.2.1).
   The approach generates permissions contracts at a higher-level of abstractions and without using any intermediate representations or method-level specifications. The inference of access permissions at the field level further shows the effectiveness of the proposed approach in enabling implicit concurrency at the more granular level. The generated specifications can be used to parallelize the execution of sequential programs written in the mainstream programming languages such as Java. The permission inference approach in this work, although focused on Java language only, should also be applicable to other object-oriented programming languages.

2. **A permission inference and verification framework**, `Sip4J`, implemented as a Java Eclipse plugin*, that automatically infers access permission contracts for single-threaded Java programs. It automatically verifies the correctness of the inferred specifications by integrating the existing permission-based model checking tool, Pulse, and performs a comprehensive concurrency analysis of the input program by extending the Pulse tool, to consider all possible side effects based on the inferred contracts (**RQ2 &**

*https://github.com/Sip4J/Sip4J

RQ3).

The `Sip4J` framework (Chapter 4) automatically generates the Plural specifications for a Java program which shows its benefits to the existing permission-based approaches such as Plural and Pulse itself, to perform their intended task without posing extra work on programmers. Moreover, the use of flow-insensitive approach to identify method's side-effects and to discover their concurrent execution outweighs other approaches that are based on the flow-sensitive analysis that does not always increase precision of the information discovered. Further, the approach computes the potential for concurrency in a sequential program by exploiting the model-checking power of the Pulse tool (Section 2.2.2). This information can be used to parallelize the execution of Java programs without the fear of data races. Furthermore, the tool automatically generates a user-friendly (Pdf visualization) report describing the behavior of the underlying program in terms of its correctness and concurrency analysis. The generated documentation can be used by both novice and expert programmers (verifiers), with equal ease, to analyze program behavior and take design decisions without having any code inspections.

3. **Empirical evaluation** of the permission inference approach and the Sip4J framework, itself in terms of its scalability, efficacy and effectiveness analysis for realistic Java applications and benchmark suites (**RQ 4 & RQ5**).

   Experiments were performed on 21 programs with 3,111 methods from four benchmark suites, widely-used in the permission-based approaches, Java Grande, Æminium, Plaid, Crystal and the Pulse tool itself to evaluate the Sip4J framework.
   The results of the evaluation (Chapter 5) have shown that the framework is indeed capable of inferring the required or safe permissions for the realistic Java programs in a correct and efficient way, without any specifications errors. The concurrency analysis of the inferred specifications has shown the effectiveness of the inferred specifications and the proposed technique itself in enabling the potential concurrency (up to 61%) from sequential programs. Further, the empirical analysis has shown that our inference technique provides reasonable support in avoiding the annotation overhead that is otherwise laborious. Furthermore, the results have shown that the framework is capable of inferring the annotations efficiently, averaging 2 seconds for annotating a single method.
   The evaluation further reveals that `Sip4J` framework is also able to identify program behavior in terms of its code reachability and null-pointer analysis that can help programmers to find syntactical errors and dead code in the program without actually compiling the code.

Moreover, our experiments have also revealed some limitations of the Pulse tool itself, in terms of its concurrency analysis to consider all possible side effects based on the inferred specifications and its support for the Java Specification Language constructs. For the first

case, we have extended the concurrency analysis in the Pulse tool as explained in Section 4.3 but for the second case, we planned to extend its analysis to incorporate more Java language constructs.
However, all the experiments demonstrated the feasibility and benefits of our permission inference approach and the framework to the existing permission-based verification such as Plural and Pulse, to perform their intended tasks without posing extra work on programmers. We believe that the inferred permissions can be used by other programming paradigms such as Æminium and Plaid, that capture the concepts of typestates, permissions, and concurrency, to perform their intended task without posing extra work on programmers and alternatively, to employ these approaches for general-purpose program development and verification purpose.

7.2 Future Work

The permission inference framework, `Sip4J`, by inferring access permissions contracts from the source code and by making their automatic analysis through the state-of-the-art model checking tool, Pulse, can help programmers a) identify some of the syntactical errors in the program such as *null-pointer* references without actually compiling the program, b) verify certain aspects of a program behavior such as code reachability analysis without performing any code inspection, and c) reason about the concurrent behavior of sequential programs as elaborated in Section 4.4 and demonstrated using realistic Java programs in Chapter 5.

Our experience with the inference of access permissions suggests that the inference of access permission contracts can further be used to automatically compute the dependencies between methods while making the side effects explicit. As access permissions pose their own ordering constraints, the computed dependencies can be used to define and automatically infer the synchronization primitives (locking and ordering constraints) from the source code of a sequential program. The generated specifications can then be used to enforce the locking policy to different program parts at different levels of granularity (method, instruction or task). The permission-based locking policy can automatically parallelize program execution for the mainstream programming languages such as Java, to the extent permitted by the computed dependencies, without using any new programming language and runtime system to support access permissions. The permission-based parallelization can free the programmers from the low-level synchronization and reasoning overhead associated with handling multi-threading in sequential programming paradigms.

In addition to supporting the race-free sharing of the heap or non-heap locations in sequential programs, the inference of permission-based synchronization constructs (such as acquire and release locks with permission invariants) can be used to verify the behavior of concurrent programs, that have already been written using multi-threading, without imposing extra work on the programmer side. The general idea is to define different pre- and post-permissions for the `acquire` and `release` locks and associate the permission information with the program reference (thread) to access a particular memory location and track the permission flow through the system in a way, that no two threads can enter simultaneously in a critical section and only

`acquired` lock can be released (pre-condition), to enforce the mutual exclusion mechanisms.

In summary, the inference of permission contracts can be used to automatically enforce mutual exclusion mechanisms in sequential programs, written in imperative programming languages such as Java, and to parallelize their execution without using the explicit concurrency constructs such as multi-threading and without imposing extra work on programmers.

To this end, we have envisaged a number of future directions relevant to the proposed permission inference framework. We plan to

a) extend the inter-procedural static analysis and combine it with dynamic analysis of the source code to incorporate more and complex Java language constructs such as polymorphism, generics, lambda expressions, and others.

b) infer access permissions at a more granular level, such as individual permissions for the members of a collection or array.

c) develop an online system that can automatically crawl a code base and generate access permissions, to encourage the wider adoption of the proposed technique,

d) automatically infer the permission-based locking and ordering constraints to develop by-default concurrent applications.

e) extend the Pulse analysis to overcome its current limitations and provide a comprehensive support for Java.

f) integrate the existing permission-based verification tool such as Plural with the permission inference framework `Sip4J`, to develop a comprehensive fully automated program verification infrastructure, for mainstream programming languages such as Java.

Research Paper Produced under this P.hD

Ayesha Sadiq, Yuan-Fang Li, Sea Ling, and Ijaz Ahmed. Extracting Permission-Based Specifications from a Sequential Java Program. In *21st International Conference on Engineering of Complex Computer Systems, United Arab Emirates, November 6-8, 2016*, pages 215–218, 2016 (**published**)

Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, page 110450, 2019b. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110450. URL http://www.sciencedirect.com/science/article/pii/S0164121219302249 (**published**)

Ayesha Sadiq, Yuan-Fang Li, Li Li, Sea Ling, and Ijaz Ahmed. Automatic inference of Symbolic Permissions for Sequential Java Programs. *Information and Software Technology*, 2019a (**under-review**)

Ayesha Sadiq, Li Li, Yuan-Fang Li, Ijaz Ahmed, and Sea Ling. Sip4J: Statically Inferring Access Permission Contracts for Parallelising Sequential Java Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, (ASE'19 Tool Demonstrations)* (**published**)

References

Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 3 2006. ISSN 0164-0925.

Abbas I. Abdel Karim. The stability of the fourth order runge-kutta method for the solution of systems of differential equations. *Commun. ACM*, 9(2):113–116, February 1966. ISSN 0001-0782. doi: 10.1145/365170.365213. URL http://doi.acm.org/10.1145/365170.365213.

Ijaz Ahmed and Néstor Cataño. Checking JML-encoded finite state machine properties. In *2018 International Conference on Advancements in Computational Sciences (ICACS)*, pages 1–9, 2 2018.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022. ACM, 2009.

Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and others. Plaid: a permission-based programming language. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 183–184. ACM, 2011.

Jonathan Aldrich, Nels E Beckman, Robert Bocchino, Karl Naden, Darpan Saini, Sven Stork, and Joshua Sunshine. The Plaid language: Typed core specification. Technical report, DTIC Document, 2012.

Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The {VerCors} Project: Setting Up Basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, 2012.

Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 172–216. Springer International Publishing, 2014.

Afshin Amighi, Christian Haack, Marieke Huisman, and Clément Hurlin. Permission-based

separation logic for multithreaded java programs. *Logical Methods in Computer Science*, 2015. ISSN 18605974.

Andrew W. Appel and Sandrine Blazy. Separation Logic for Small-Step cminor. In *Theorem Proving in Higher Order Logics*, pages 5–21. Springer Berlin Heidelberg, 2007.

Wladimir Araujo, Lionel Briand, and Yvan Labiche. Concurrent contracts for Java in JML. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 37–46. IEEE, 2008.

Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Software Testing Verification and Reliability*, 2003.

Bernhard. Beckert, Reiner. Hähnle, and P. H. (Peter H.) Schmitt. *Verification of object-oriented software : the KeY approach.* Springer, 2007.

Nels E Beckman. Modular typestate checking in concurrent Java programs. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 737–738. ACM, 2009.

Nels E Beckman. *Types for Correct Concurrent API Usage, PhD thesis, technical report CMU-ISR-10-131.* PhD thesis, 12 2010.

Nels E Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying Correct Usage of Atomic Blocks and Typestate. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 227–244. ACM, 2008.

Nels E Beckman, Duri Kim, and Jonathan Aldrich. An Empirical Study of Object Protocols in the Wild. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 2–26. Springer-Verlag, 2011.

Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, Paul Thomson, Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, volume 47, page 113. ACM Press, 2012.

Kevin Bierhoff. Iterator Specification with Typestates. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 79–82. ACM, 2006.

Kevin Bierhoff. Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 19–32. ACM, 2011.

Kevin Bierhoff and Jonathan Aldrich. *Modular typestate checking of aliased objects*, volume 42 of *OOPSLA '07*. ACM, 2007.

Kevin Bierhoff and Jonathan Aldrich. PLURAL: Checking Protocol Compliance Under Aliasing. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 971–972. ACM, 2008.

Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Polymorphic fractional permission inference. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '09, 2009a.

Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Practical API Protocol Checking with Access Permissions. In *ECOOP*, pages 195–219, 2009b.

Bierhoff, Kevin. Api protocol compliance in object-oriented software. 2009.

Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

Stefan Blom, Marieke Huisman, and Matej Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 2014. ISSN 01676423.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission Accounting in Separation Logic. *SIGPLAN Not.*, 40(1):259–270, 1 2005. ISSN 0362-1340.

Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 56–69. ACM, 2001.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230. ACM, 2002.

John Boyland. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72. Springer-Verlag, 2003.

John Boyland. Why we should not add readonly to Java (yet). *The Journal of Object Technology*, 5(5):5, 2006. ISSN 1660-1769.

John Tang Boyland. Semantics of Fractional Permissions with Nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, 8 2010. ISSN 0164-0925.

John Tang Boyland, Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Constraint Semantics for Abstract Read Permissions. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs - FTfJP'14*, pages 1–6. ACM Press, 2014.

Stephen Brookes. A Semantics for Concurrent Separation Logic. pages 16–34. Springer, Berlin, Heidelberg, 2004.

J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/1096-9128.

David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

C. Barrett A. Stump and C Tinelli. The SMTLIB Standard, Version 2.0, 2010.

Luis Caires. Spatial-behavioral Types for Concurrency and Resource Control in Distributed Systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 7 2008. ISSN 0304-3975.

Luis Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part II). In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 209–225. Springer-Verlag, 2002.

Luis Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part I). *Inf. Comput.*, 186(2):194–235, 11 2003. ISSN 0890-5401.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

Luis Caires and João C Seco. The Type Discipline of Behavioral Separation. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 275–286. ACM, 2013.

Andrea Capriccioli, Marco Servetto, and Elena Zucca. An Imperative Pure Calculus. *Electron. Notes Theor. Comput. Sci.*, 322(C):87–102, 4 2016. ISSN 1571-0661.

S. A. Carr, F. Logozzo, and M. Payer. Automatic contract insertion with ccbot. *IEEE Transactions on Software Engineering*, 43(8):701–714, Aug 2017.

Néstor Cataño, Ijaz Ahmed, Radu I Siminiceanu, and Jonathan Aldrich. A case study on the lightweight verification of a multi-threaded task server. *Sci. Comput. Program.*, 80: 169–187, 2014.

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61. ACM, 2011.

S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 6 2004. ISSN 0098-5589.

Sagar Chaki and Arie Gurfinkel. BDD-Based Symbolic Model Checking. In *Handbook of Model Checking*, pages 219–245. Springer International Publishing, 2018.

Sagar Chaki, Sriram K Rajamani, and Jakob Rehof. Types As Models: Model Checking Message-passing Programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 45–57. ACM, 2002.

Dave Clarke and Tobias Wrigstad. External Uniqueness Is Unique Enough. pages 176–200. Springer, Berlin, Heidelberg, 2003.

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. pages 15–58. Springer, Berlin, Heidelberg, 2013.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy Mcneil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th . . .*, 2015.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. pages 23–42. Springer, Berlin, Heidelberg, 2009.

David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011.

Patrick Cousot. Abstract Interpretation. *ACM Comput. Surv.*, 28(2):324–328, 6 1996. ISSN 0360-0300.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

Samir R. Das and Richard M. Fujimoto. A performance study of the cancelback protocol for time warp. *SIGSIM Simul. Dig.*, 23(1):135–142, July 1993. ISSN 0163-6103. doi: 10.1145/174134.158476. URL http://doi.acm.org/10.1145/174134.158476.

Robert DeLine and Manuel Fähndrich. Adoption and focus: Practical linear types for imperative programming. *Programming language design and implementation, ACM SIGPLAN*, 2002. ISSN 0362-1340.

Robert DeLine and Manuel Fähndrich. Typestates for Objects. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 465–490. Springer Berlin Heidelberg, 2004.

Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2005.

Ricardo J. Dias, Vasco Pessanha, and João M. Lourenço. Precise Detection of Atomicity Violations. pages 8–23. Springer, Berlin, Heidelberg, 2013.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528. Springer-Verlag, 2010.

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009.

Jérôme Dohrau, Alexander J. Summers, Caterina Urban, Severin Münger, and Peter Müller. Permission inference for array programs. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 55–74, Cham, 2018. Springer International Publishing.

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A Data-centric Approach to Synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4:1–4:48, 5 2012. ISSN 0164-0925.

Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3):22–44, June 1992. ISSN 0163-5964. doi: 10.1145/141868.141871. URL http://doi.acm.org/10.1145/141868.141871.

Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0. doi: 10.1109/ICPP.2009.64. URL https://doi.org/10.1109/ICPP.2009.64.

Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252. ACM, 2003.

Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 10–30. Springer-Verlag, 2011.

Pietro Ferrara and Peter Müller. Automatic Inference of Access Permissions. In *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 202–218. Springer Berlin Heidelberg, 2012.

Jean Christophe Filliâtre and Andrei Paskevich. Why3 - Where programs meet provers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013.

Cormac Flanagan, Shaz Qadeer, Cormac Flanagan, and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03*, volume 38, pages 338–349. ACM Press, 2003.

Cormac Flanagan, Stephen N Freund, Marina Lifshin, and Shaz Qadeer. Types for Atomicity: Static Checking and Inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, 8 2008. ISSN 0164-0925.

Robert W. Floyd. Assigning Meanings to Programs. pages 65–81. Springer, Dordrecht, 1993.

Alcides Fonseca and Bruno Cabral. ÆminiumGPU: An Intelligent Framework for GPU Programming. In *Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing*, pages 96–107. Springer Berlin Heidelberg, 2013.

Alcides Fonseca, Bruno Cabral, João Rafael, and Ivo Correia. Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime. *International Journal of Parallel Programming*, 44(6):1337–1358, 2016.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 10 2014. ISSN 0164-0925.

Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 2010. ISSN 09567968.

Simon J. Gay, Nils Gesbert, António Ravara, and Vasco T. Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 2015a. ISSN 18605974.

Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015b. doi: 10.2168/LMCS-11(4:12)2015. URL https://doi.org/10.2168/LMCS-11(4:12)2015.

Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM 2011: Formal Methods*, pages 386–401. Springer Berlin Heidelberg, 2011.

Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca. Tracing sharing in an imperative pure calculus. *CoRR*, abs/1803.0, 2018a.

Paola Giannini, Marco Servetto, and Elena Zucca. A Type and Effect System for Uniqueness and Immutability. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1038–1045. ACM, 2018b.

Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the*

ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12, 2012.

Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular Shape Analysis. *SIGPLAN Not.*, 42(6):266–277, 6 2007. ISSN 0362-1340.

Aaron Greenhouse. A Programmer-Oriented Approach to Safe Concurrency. Technical report, 2003. URL http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-135.pdf.

Aaron Greenhouse and William L Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001. ISSN 0164-0925. doi: 10.1145/506315.506316. URL http://doi.acm.org/10.1145/506315.506316.

Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69849-4.

Christian Haack and Clément Hurlin. Separation Logic Contracts for a Java-Like Language with Fork/Join. In *Algebraic Methodology and Software Technology*, pages 199–215. Springer Berlin Heidelberg, 2008.

Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.

Christian Hammer, J Dolby, M Vaziri, and F Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *Proceedigns of the 30th International Conference on Software Engineering (ICSE' 08)*, 2008.

Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190075.

Stefan Heule, K Rustan M Leino, Peter Müller, and Alexander J Summers. Fractional permissions without the fractions. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, page 1. ACM, 2011.

Stefan Heule, K Rustan M Leino, Peter Müller, and Alexander J Summers. Abstract read permissions: Fractional permissions without the fractions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–334. Springer, 2013.

C. A. R. Hoare. Towards a Theory of Parallel Programming. In *The Origin of Concurrent Programming*, pages 231–244. Springer New York, 1972.

C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 1974. ISSN 00010782.

Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 2012. ISSN 18605974.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.

Kohei Honda. Types for dyadic interaction. pages 509–523. Springer, Berlin, Heidelberg, 1993.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138. Springer-Verlag, 1998.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284. ACM, 2008.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

J Hughes. Why Functional Programming Matters. *Comput. J.*, 32(2):98–107, 4 1989. ISSN 0010-4620.

Marieke Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle. PhD Thesis.* PhD thesis, Computing Science Institute, University of Nijmegen, 2001.

Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *Proceedings - IEEE 14th International Symposium on Parallel and Distributed Computing, ISPDC 2015*, 2015.

Michael R. A. Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-65602-8.

Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 3 2005. ISSN 0164-0925.

B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 137–146. IEEE, 2005.

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 271–282. ACM, 2011.

Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. pages 304–311. Springer, Berlin, Heidelberg, 11 2010. doi: 10.1007/978-3-642-17164-2{_}21. URL http://link.springer.com/10.1007/978-3-642-17164-2_21.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification of single-threaded and multithreaded programs. *ACM TOPLAS*, 40(3), 2018.

Jonas Braband Jensen and Lars Birkedal. Fictional Separation Logic. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 377–396. Springer Berlin Heidelberg, 2012.

Cliff B Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, 1983.

Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers. Viper. Technical report, 2014.

Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009. ISBN 0849328802, 9780849328800.

Nicholas Kidd, Thomas Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. *International Journal on Software Tools for Technology Transfer*, 2011. ISSN 14332779.

Taekgoo Kim, Kevin Bierhoff, Jonathan Aldrich, and Sungwon Kang. Typestate protocol specification in JML. In *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, pages 11–18. ACM, 2009.

Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Transactions on Programming Languages and Systems*, 32(5):1–49, 5 2010. ISSN 01640925.

N. Shankar L. de Moura, S. Owre. The SAL Language Manual, Tech. Rep. SRI-CSL-01-02. Technical report, CSL Technical Report, 2003.

Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 17–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2010.

Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. Variable Permissions for Concurrency Verification. In *Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, pages 5–21. Springer Berlin Heidelberg, 2012.

Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465. URL http://doi.acm.org/10.1145/337449.337465.

Gary T Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009.

K Rustan M Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. *ACM SIGPLAN Notices*, 37(5):246–257, 2002.

K Rustan M Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, pages 195–222. Springer Berlin Heidelberg, 2009.

Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.

Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In Görel Hedin, editor, *Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36579-2.

Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes. *ACM SIGOPS Operating Systems Review*, 42(2):329, 3 2008. ISSN 01635980.

D Marino, C Hammer, J Dolby, M Vaziri, F Tip, and J Vitek. Detecting deadlock in programs with data-centric synchronization. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 322–331, 5 2013.

Nicholas D. Matsakis, Felix S. Klock, Nicholas D. Matsakis, and Felix S. Klock II. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14*, volume 34, pages 103–104. ACM Press, 2014.

B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 10 1992. ISSN 0018-9162.

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

Filipe Militao, Jonathan Aldrich, and Luis Caires. Aliasing Control with View-based Typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 7:1–7:7. ACM, 2010.

Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee Protocols. In *ECOOP 2014 – Object-Oriented Programming*, pages 334–359. Springer Berlin Heidelberg, 2014a.

Filipe Militão, Jonathan Aldrich, and Luis Caires. Substructural Typestates. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification*, PLPV '14, pages 15–26. ACM, 2014b.

Filipe Militão, Jonathan Aldrich, and Luís Caires. Composing interfering abstract: Protocols. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, volume 56, pages 161–1626, 2016.

Filipe Milito and Luis Caires. An Exception Aware Behavioral Type System for Object-Oriented Programs. In *INFORUM 2009 - Simpósio de Informática*, pages 1–12. Faculdade de Ciências - Universidade de Lisboa, 2009.

Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications*, pages 293–307. Springer Berlin Heidelberg, 2005.

Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, 2007.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*. 2017.

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A Type System for Borrowing Permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 557–570. ACM, 2012.

Mayur Naik, Chang Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings - International Conference on Software Engineering*, 2009.

Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 355–369. Springer-Verlag, 2008.

Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. pages 493–508. Springer, Berlin, Heidelberg, 1993.

Flemming Nielson and Hanne Riis Nielson. From CML to its process algebra. *Theoretical Computer Science*, 1996. ISSN 03043975.

James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1998.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic*, pages 1–19. Springer Berlin Heidelberg, 2001.

Peter W OHearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 4 2007. ISSN 0304-3975.

Susan Owicki and David Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM*, 19(5):279–285, 5 1976. ISSN 0001-0782.

S Owre, J M Rushby, and N Shankar. PVS: a prototype verification system. *11th International Conference on Automated Deduction*, 1992. ISSN 16113349.

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *ACM SIGPLAN Notices*, 40(1):247–258, 2005. ISSN 03621340.

Hervé Paulino, Daniel Parreira, Nuno Delgado, António Ravara, and Ana Matos. From Atomic Variables to Data-centric Concurrency Control. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1806–1811. ACM, 2016.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings - International Conference on Software Engineering*, 2012.

João Rafael, Ivo Correia, Alcides Fonseca, and Bruno Cabral. Dependency-based automatic parallelization of java applications. In *European Conference on Parallel Processing*, pages 182–193. Springer, 2014.

John C Reynolds. Syntactic Control of Interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 39–46. ACM, 1978.

John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, GaryT. Leavens, and Robby. Extending JML for Modular Specification and Verification of Multi-threaded Programs. In *ECOOP 2005 - Object-Oriented Programming*. 2005.

Pierre Roux and Radu Siminiceanu. Model Checking with Edge-valued Decision Diagrams. In *Second {NASA} Formal Methods Symposium - {NFM} 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*, pages 222–226, 2010.

Ayesha Sadiq, Li Li, Yuan-Fang Li, Ijaz Ahmed, and Sea Ling. Sip4J: Statically Inferring Access Permission Contracts for Parallelising Sequential Java Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, (ASE'19 Tool Demonstrations)*.

Ayesha Sadiq, Yuan-Fang Li, Sea Ling, and Ijaz Ahmed. Extracting Permission-Based Specifications from a Sequential Java Program. In *21st International Conference on Engineering of Complex Computer Systems, United Arab Emirates, November 6-8, 2016*, pages 215–218, 2016.

Ayesha Sadiq, Yuan-Fang Li, Li Li, Sea Ling, and Ijaz Ahmed. Automatic inference of Symbolic Permissions for Sequential Java Programs. *Information and Software Technology*, 2019a.

Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, page 110450, 2019b. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110450. URL http://www.sciencedirect.com/science/article/pii/S0164121219302249.

Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 1999. ISSN 03043975.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare Triples and Frame Rules for Higher-order Store. *Logical Methods in Computer Science*, 7(3), 2011.

Radu I Siminiceanu, Ijaz Ahmed, and Néstor Cataño. Automated Verification of Specifications with Typestates and Access Permissions. *ECEASST*, 53, 2012.

Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. pages 148–172. Springer, Berlin, Heidelberg, 2009.

Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *OOPSLA Companion*, pages 933–940, 2009.

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. aem: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.*, 36(1):1–42, 2014. ISSN 0164-0925.

Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. *ACM SIGPLAN Notices*, 2011. ISSN 03621340.

Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7): 54–62, 9 2005. ISSN 1542-7730.

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *In PARLE'94, volume 817 of LNCS*, pages 398–413, 1994.

Viktor Vafeiadis and Matthew Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In Luís Caires and Vasco T Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271. Springer Berlin Heidelberg, 2007.

Mandana Vaziri, Frank Tip, and Julian Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345. ACM, 2006.

Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A Type System for Data-Centric Synchronization. In *ECOOP 2010 – Object-Oriented Programming*, pages 304–328. Springer Berlin Heidelberg, 2010.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps That Hop with Heap-Hop. pages 275–279. Springer, Berlin, Heidelberg, 2010.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003. ISSN 09288910.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 205. ACM Press, 2007.

Philip Wadler. Propositions as sessions. In *Journal of Functional Programming*, 2014.

Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical Permissions for Race-free Parallelism. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 614–639. Springer-Verlag, 2012.

John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010. ISSN 03029743.

Min Xu, Rastislav Bodík, Mark D. Hill, Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *ACM SIGPLAN Notices*, 40(6):1, 6 2005. ISSN 03621340.

Yang Zhao. *Concurrency analysis based on fractional permissions.* PhD thesis, University of Wisconsin-Milwaukee, 2007.

Yang Zhao, Ligong Yu, and Jia Bei. The Permission Approach to Comprehend Lock-Based Synchronization Policy. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 709–713. IEEE, 12 2008.

Appendices

Appendix 1

Syntactic Rules for Modelling Object Accesses

A.1 Modelling Object Accesses in the Current Method

This section presents the syntactic rules that capture expression statements in a method and that have been used to support the data-flow and alias-flow analysis of the source code during the metadata extraction phase of the permission inference approach, as explained in Section 3.2.1. The rules are being divided into two main types depending on their access by the current method and other methods: a) `Statement` rules, and b) `Context` rules, during the analysis. The rules are further categorized according to the type of expressions such as `FieldAccess`, `Assignment` or `MethodInvocation`, encountered during each expression statement and the type of reference variable (`<grv>`, `<lrv>`, `<lv>`) accessed in each expression. During parsing, each expression is recursively parsed to fetch the type of expressions designed as based cases. The extracted dependencies are then mapped in the form of a permission-based graph model as explained previously in Section 3.2.2.

The rules follow the style of sequent calculus in Linear logic, with connectives and implication ($\multimap$) operator, to enforce the strong and constructive interpretation of the specified rules as formulas, to support the analysis.

A complete list of the modelling rules is given below.

I. Statement Rules for the Global References

$$\frac{\texttt{<Type> <grv>}}{\texttt{<do-nothing>}}\ (\textbf{GR-Decl, <grv>})$$

$$\frac{\texttt{<grv> |super.<grv> |this.<grv> |<ClassName>.<grv> |<obj>.<grv>}}{\texttt{addReadEdge(this\_m, <grv>)}}\ (\textbf{GR-Read-Only, <grv>})$$

$$\frac{\texttt{[PRIM\_TYPE] <grv> = <grv1> |<LITERAL>}}{\texttt{addWriteEdge(this\_m, <grv>)}(\forall a \in \texttt{aliasOf(<grv>) addWriteEdge(this\_m, a)),(apply(GR-Read-Only, <grv1>)}}\ (\textbf{GR-Val-Flow,<grv>})$$

$$\frac{\texttt{[<Type>] <grv> = new <Type>(<grv2>)|<Number\_Literal>}}{\texttt{(addWriteEdge(this\_m,<grv>)} \multimap \texttt{apply(Context-N,<grv>)),(apply(GR-Read-Only, <grv2>)}}\ (\textbf{GR-New-Obj, <grv>})$$

$$\frac{\texttt{[<REF\_TYPE>] <grv> = <grv1>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<grv>, <grv2>)} \multimap \texttt{removeAliasEdge(<grv>, <grv2>)), addAliasEdge(<grv>, <grv1>)}\\ \texttt{apply(GR-Read-Only, <grv1>)}\end{array}}\ (\textbf{GR-Add-Flow, <grv>})$$

$$\frac{\texttt{[<REF\_TYPE>] <grv> = <lrv>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv1>)} \multimap \texttt{addAliasEdge(<grv>, <lrv>))}\\ \\ \texttt{(}\exists\texttt{aliasEdge(<grv>, <grv2>)} \multimap \texttt{removeAliasEdge(<grv>, <grv2>)),apply(GR-Read-Only, <grv1>)}\end{array}}\ (\textbf{GR-Addr-Flow, <lrv>})$$

$$\frac{\texttt{[<REF\_TYPE>] <grv> = <lv>}}{\texttt{(}\exists\texttt{aliasEdge(<grv>, <grv1>)} \multimap \texttt{removeAliasEdge(<grv>, <grv1>))}}\ (\textbf{GR-Addr-Flow, <lv>})$$

$$\frac{\texttt{<Type> <grv> = <Null\_Literal>|MCall(<post-perm>,<grv1>)}}{\texttt{<do-nothing>|MCall(<post-perm>,<grv1>)}}\ (\textbf{GR-NullAddr-Init, <grv>})$$

$$\frac{\texttt{<grv> = <Null\_Literal>|MCall(<post-perm>,<grv2>)}}{\begin{array}{l}\texttt{addWriteEdge(this\_m, <grv>)(}\exists\texttt{aliasEdge(<grv>, <grv1>)} \multimap \texttt{removeAliasEdge(<grv>, <grv1>)}\\ \texttt{apply(Context-N, <grv>)),(}\forall\ \texttt{a} \in \texttt{aliasOf(<grv>),apply(<GR-NullAddr-Flow>, a)))}\\ \texttt{,apply(MCall(<post-perm>, <grv1>))}\end{array}}\ (\textbf{GR-NullAddr-Flow, <grv>})$$

$$\frac{\texttt{<grv> = <lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \multimap \texttt{<do-nothing>)}}\ (\textbf{GR-SelfAddr-Flow, <lrv>})$$

$$\frac{\texttt{<grv> = <grv>}}{\texttt{<do-nothing>}}\ (\textbf{GR-SelfAddr-Flow, <grv>})$$

II. Statement Rules for the Local References

$$\frac{\texttt{<lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)}\multimap\texttt{(apply(GR-Read-Only(<grv>))}} \text{ (LR-Read-Only, <lrv>)}$$

$$\frac{\texttt{<lrv>.<grv1> = <Number\_Literal>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)}\multimap\texttt{apply(GR-Val-Flow, <grv1>))}} \text{ (LR-Val-Flow, <lrv>)}$$

$$\frac{\texttt{[<REF\_TYPE>] <lrv> = <grv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv1>)}\multimap\texttt{removeAliasEdge(<lrv>,<grv1>)),addAliasEdge(<lrv>,<grv>))}\quad\texttt{apply(GR-Read-Only, <grv>)}} \text{ (LR-Addr-Flow, <grv>)}$$

$$\frac{\texttt{[<REF\_TYPE>] <lrv> = <lrv1>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>,<grv>)}\multimap\texttt{removeAliasEdge(<lrv>,<grv>)),}\quad\texttt{(}\exists\texttt{aliasEdge(<lrv1>,<grv1>)}\multimap\texttt{addAliasEdge(<lrv>,<lrv1>),apply(GR-Read-Only, <grv1>))}} \text{ (LR-Addr-Flow, <lrv>)}$$

$$\frac{\texttt{[<REF\_TYPE>] <lrv> = <lv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)}\multimap\texttt{removeAliasEdge(<lrv>,<grv>))}} \text{ (LR-Addr-Flow, <lv>)}$$

$$\frac{\texttt{<lrv> = new <[<REF\_TYPE>]>(<grv2> |<Number\_Literal>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>,<grv1>)}\multimap\texttt{removeAliasEdge(<lrv>,<grv1>)}\quad\texttt{(apply(GR-Read-Only,<grv2>))}} \text{ (LR-New-Obj, <lrv>)}$$

$$\frac{\texttt{<lrv> = <Null\_Literal>|MCall(<post-perm>,<grv>)}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)}\multimap\texttt{removeAliasEdge(<lrv>, <grv>),}\quad\texttt{(}\forall\ \texttt{a}\in\texttt{aliasOf(<lrv>),apply(<GR-NullAddr-Flow>, a)),apply(MCall(<post-perm>,<grv>))}} \text{ (LR-NullAddr-Flow, <lrv>)}$$

$$\frac{\texttt{<Type> <lrv> = <Null\_Literal>|MCall(<post-perm>,<grv>)}}{\texttt{<do-nothing>)|MCall(<post-perm>,<grv>)}} \text{ (LR-NullAddr-Init, <lrv>)}$$

$$\frac{\texttt{<lrv> = <grv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>,<grv>)}\multimap\texttt{<do-nothing>}} \text{ (LR-SelfAddr-Flow, <lrv>)}$$

$$\frac{\texttt{<lrv1> = <lrv1>}}{\texttt{<do-nothing>}} \text{ (LR-SelfAddr-Flow, <lrv>)}$$

III. Statement Rules for Method Calls

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addReadEdge(<grv>, this\_m) apply(Context-R, <grv>)}}\ \textbf{MCall(Immutable, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addReadEdge(this\_m, <grv>), apply(Context-RW, <grv>)}}\ \textbf{MCall(Pure, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addWriteEdge(this\_m, <grv>), apply(Context-R, <grv>)}}\ \textbf{MCall(Full, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addWriteEdge(this\_m, <grv>) apply(Context-RW, <grv>)}}\ \textbf{MCall(Share, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{addWriteEdge(this\_m, <grv>), apply(Context-N, <grv>)}}\ \textbf{MCall(Unique, <grv>)}$$

$$\frac{\texttt{MCall([<args>])|super.MCall([<args>])|super([<args>])}}{\texttt{<do-nothing>}}\ \textbf{MCall(None, <grv>)}$$

A.2 Modelling Object Accesses through Other Methods

As discussed previously in Section 3.2.2, the context rules model the read, write behavior of other methods on the shared objects accessed in the current method.

I. **Context Rules for the Global References**

$$\frac{\texttt{<grv>}}{\texttt{addReadEdge(context, <grv>)}}\ \textbf{(Context-R, <grv>)}$$

$$\frac{\texttt{<grv>}}{\texttt{addReadEdge(context, <grv>), addWriteEdge(context, <grv>)}}\ \textbf{(Context-RW, <grv>)}$$

$$\frac{\texttt{<grv>}}{\texttt{removeReadEdge(context, <grv>), removeWriteEdge(context, <grv>)}}\ \textbf{(Context-N, <grv>)}$$

II. **Context Rules for the Local References**

$$\frac{\texttt{<lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \multimap \texttt{apply(Context-R, <grv>))}}\ \textbf{(Context-R, <grv>)}$$

$$\frac{\texttt{<lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \multimap \texttt{apply(Context-RW, <grv>))}}\ \textbf{(Context-RW, <grv>)}$$

$$\frac{\texttt{<lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \multimap \texttt{apply(Context-N, <grv>))}}\ \textbf{(Context-N, <grv>)}$$

A.3 Access Permission Inference Rules

This section lists the access permission inference rules used to generate five types of symbolic permissions on the objects referenced in a method, as explained previously in Section 3.2.3.

$$\frac{\neg\exists\texttt{readEdge(context,<grv>)} \wedge \neg\exists\texttt{writeEdge(context, <grv>)}\exists\texttt{readEdge(this\_m, <grv>)} \wedge \exists\texttt{writeEdge(this\_m, <grv>)}}{\texttt{unique(<grv>)}}\ (\textbf{Unique})$$

$$\frac{\neg\exists\texttt{writeEdge(this\_m,<grv>)} \wedge \exists\texttt{readEdge(<grv>,this\_m)} \wedge \neg\exists\texttt{writeEdge(context,<grv>)} \wedge \exists\texttt{readEdge(<grv>,context)}}{\texttt{immutable(<grv>)}}\ (\textbf{Immutable})$$

$$\frac{\exists\texttt{readWriteEdge(this\_m,<grv>)} \wedge \neg\exists\texttt{writeEdge(context,<grv>)} \wedge \exists\texttt{readEdge(<grv>,context)}}{\texttt{full(<grv>)}}\ (\textbf{Full})$$

$$\frac{\exists\texttt{writeEdge(this\_m,<grv>)} \wedge \exists\texttt{readEdge(<grv>,this\_m)} \wedge \exists\texttt{writeEdge(context,<grv>)} \wedge \exists\texttt{readEdge(<grv>, context)}}{\texttt{share(<grv>)}}\ (\textbf{Share})$$

$$\frac{\neg\exists\texttt{writeEdge(this\_m,<grv>)} \wedge \exists\texttt{readEdge(<grv>,context)} \wedge \exists\texttt{writeEdge(context,<grv>)}}{\texttt{pure(<grv>)}}\ (\textbf{Pure})$$

$$\frac{\neg\exists\texttt{writeEdge(this\_m,<grv>)} \wedge\exists\texttt{readEdge(<grv>,this\_m)} \wedge\neg\exists\texttt{writeEdge(context,<grv>)} \wedge\neg\exists\texttt{readEdge(<grv>,context)}}{\texttt{none(<grv>)}}\ (\textbf{None})$$

Appendix 2

Computational Complexity Analysis of the Permission Inference Approach

This section presents the computational complexity analysis of the permission inference approach presented in this thesis. The inference approach consists of three main tasks: code parsing, graph construction and graph traversal. Therefore, the computation complexity analysis of the inference approach depends on these three tasks.

For a Java program with C number of classes and M number of methods, the computation complexity ($CA\text{-}Perm(M)$) for the first task (code parsing) depends on the data-flow and alias-flow analysis ($DFAA(M)$) of M methods, to extract the read and write information of the referenced objects (RV) in each method, along with the context analysis ($CA(M)$) of M methods to fetch the read and write access of referenced objects (RV) by other methods. The computational complexity of the second phase depends on the number of computation steps to model the extracted information in the form of permission-based graph model for M methods. It further depends on the graph traversal ($GT(M)$) phase to generate access permissions on the referenced object (RV) for M methods. The computation complexity of the proposed approach depends on the following parameters.

M= No. of methods to be parsed
N= No. of method calls in a program
ES= No. of expression statements in a method
$\mathbb{M}$= No. of methods already stored in a data structure
P= No. of parameters in a method
$\mathbb{P}$= No. of parameters already stored in a data structure
RV= No. of the reference variable in a method
$\mathbb{RV}$= No. of reference variable already stored in a data structure
LRV= No. of local reference variables in a method
RA= No. of aliases against each reference variable in a method
$\mathbb{RA}$= No. of aliases already stored in a data structure
Arg(N)= No. of arguments in N method calls

The number of computation steps are shown below by detailing the tasks in each phase.

CA -Perm = DFAA(M) + CA(M) + GC(M) + GT(M)

DFAA(M) $= parseMethods(M)$

parseMethods(M) $= M * parseMethod(m)$

parseMethod(m) = createMethod(m) + parseMParams(m) + parseMStatements(m)

createMethod(m) $= \mathbb{M}$

parseMParams(m) $= P*(N*Arg(N))+\mathbb{P}+\mathbb{RV})$

parse-MStmts(m) $=$ **parse-RO-exprs(exp)** $+$ **parse-MC-exprs(exp)** $+$ **parse-ASS-exprs(exp)**

parse-MStmts(m) $= (ES*(RV($

(**parse-RO-exprs(exp)**) $= (\mathbb{P}+(\mathbb{RV}*\mathbb{RA}))\wedge$

(**parse-MC-exprs(exp)**) $= parseMethods(m)\wedge$

(**parse-ASS-exprs(exp)**)) $=$
$(2*(\mathbb{P}+(\mathbb{RV}*\mathbb{RA})))\,||(\mathbb{P}+2*(\mathbb{RV}*\mathbb{RA})+DFAA(N))\,||$
$(\mathbb{P}+2*(\mathbb{RV}*\mathbb{RA}))||(\mathbb{P}+(\mathbb{RV}*\mathbb{RA}))$
$=2*(\mathbb{P}+(\mathbb{RV}*\mathbb{RA}))+DFAA(N)$

DFAA(M) $= M*(\mathbb{M}+(P*((N*Arg(N))+\mathbb{P}+\mathbb{RV}))+(ES*(RV*(\mathbb{P}+(\mathbb{RV}*\mathbb{RA}))+2*(\mathbb{P}+(\mathbb{RV}*\mathbb{RA})))+DFAA(N)))$

CA(M) $= (M*(\mathbb{RV}^2))$

GC(M) $= (M*\mathbb{RV})$

GT(M) $= (M*\mathbb{RV})$

Hence, **CA -Perm = DFAA(M) + CA(M) + GC(M) + GT(M)** $=$
$M * ((P * (N * Arg(N))) + (ES * 2 * (RV * (\mathbb{RV} * \mathbb{RA}))) + (RV^2)) + DFAA(N))$