Sip4J: Statically inferring permission-based specifications for sequential Java programs

Ayesha Sadiq*, Yuan-Fang Li, Sea Ling, Li Li

Faculty of Information Technology, Monash University, Clayton, Australia

Ijaz Ahmed

Department of Computer and Information Sciences, Higher College Technology, Dubai, UAE

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. Given the intricacies in creating these constructs, it is very likely for a programmer to omit important dependencies (constraints) or to create wrong or misspelled specifications that may lead to problems such as race conditions and deadlocks. With these considerations in mind, access permission-based dependencies have been investigated as an alternative approach to formally verify the correctness of multi-threaded programs and to exploit the implicit concurrency present in sequential programs. 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. However, significant annotation overhead can arise from manually adding permission-based specifications in a source program, diminishing the effectiveness of existing permission-based approaches. In this paper, we present a framework, Sip4J, to automatically extract implicit dependencies from sequential Java programs in the form of *access permissions*, by performing inter-procedural static analysis of the source code. We have integrated an existing tool Pulse to automatically verify correctness of the inferred specifications and to reason about their concurrent behaviors. Our evaluation on some widely-used benchmarks gives strong evidence of the correctness of the inferred annotations and their effectiveness in enabling concurrency in sequential programs.

Keywords:
Access permissions, permission inference, static analysis, object oriented programs, Java

*Corresponding author

Email addresses: ayesha.sadiq@monash.edu (Ayesha Sadiq), yuanfang.li@monash.edu (Yuan-Fang Li), chris.ling@monash.edu (Sea Ling), li.li@monash.edu (Li Li), ijaz.uma@gmail.com (Ijaz Ahmed)

1. Introduction

Enabling concurrency for imperative and object-oriented languages has become one of the grand challenges for the IT industry today[1]. This is because of the nature of the imperative and object-oriented programming paradigms where the compiler follows the execution order in which the program is written, i.e., sequential. In such languages, programmers manually introduce concurrency by using explicit concurrency constructs, e.g., multi-threading-related classes such as *Thread*, *Runnable* in Java. Unfortunately, traditional multi-threading paradigm frequently results in deadlocks or unwarranted race conditions that are hard to debug. **Access permission**, a way to model and reason about concurrencies in software programs, is hence 10 introduced to alleviate these issues. Access permission, formally called Boyland's fractional permission, are abstract capabilities that encodes effects of read/write operations and alias information for a referenced object [1].

Symbolic permissions [2] simply called *access permissions* is an extension of fractional permissions [1] that were originally proposed to ensure the non-interferences of program states in a parallel program but instead of using fractional values to represent and split permissions among multiple references, symbolic permissions represent and track permission flow through the system using high level of abstractions (permission types) such as immutable or unique etc.

A group of CMU researchers led by Jonathan Aldrich manually wrote permission-based typestate specifications on a number of APIs to model and reason about the correctness of typestate-based sequential and concurrent programs [2–6] etc., and further parallelise the execution of these programs in a permission-based typestate paradigm Plaid [7]. Further, the group worked in a joint research project, Æminium [8] and proposed a formal language and a runtime to develop by-default concurrent applications based on access permissions. Similarly, access permission-based verification of concurrent programs and inference of access notations has recently been investigated by Peter Müller and his colleagues [9–16]. Moreover, permission-based access notations have been used in many formal approaches to address issues related to safe concurrency, security and verification of functional and domain specific properties [17–22] and many more.

Unfortunately, in order to benefit from access permissions, programmers have to manually add appropriate permission-based specifications (e.g., annotations) as dependency information in the program. Not only do programmers need to spend time getting familiarised with a completely new specification language and runtime system, they also need to manually write specifications in the source code which is laborious and error-prone. These issues have hindered the wider adoption of access permission-based approaches. To this end, in our work, we aim to resolve the aforementioned issues by introducing to the community a novel approach that infers permission-based dependencies from the source program and automatically annotate programs with permission-based specifications.

We are interested in inferring permission-based specifications for Java programs. This paper presents a comprehensive framework called `Sip4J` to infer access permission-based specifications from a sequential Java program using permission inference technique. To infer

[1] UK Computing Research Committee, Grand Challenges in Computing Research. `http://www.ukcrc.org.uk/grand-challenge/`.

permission-based dependencies from the source code, our technique is based on Abstract Syntax Tree which follows a set of syntactic rules and graph abstractions. A tool based on the technique is implemented as an Eclipse Plugin, integrated with an existing permission-based verification tool Pulse [19]. We then empirically evaluate the proposed technique by verifying the correctness of the inferred specifications and hence demonstrating the effectiveness of the tool. We also measure the execution speed of our inference technique on a number of benchmark Java programs and compare its performance with their corresponding Plaid counterparts.

It is worth mentioning that the technique proposed in this work, although focused on Java language only, should also be applicable to other object-oriented programming languages. Indeed, we believe that the inferred specifications and our framework can be used by existing permission-based verification approaches to verify the correctness of a program without annotation overhead. It is also able to discover some of the syntactical errors in a program such as `NullPointer` references at compile time. Ultimately, it can be used to reason about the concurrent behaviour of a sequential program without imposing extra work on the programmers. To help readers to replicate our results, we have provided the `Sip4J` program, the benchmark programs and their analysis reports by `Sip4J` and Pulse on GitHub: `https://github.com/Sip4J/sip4j`.

The rest of the paper is structured as follows. Section 2 provides an introduction to access permissions and an overview of the tools Pulse and the associated Plural language. Section 3 describes the `Sip4J` framework to generate and verify permission-based specifications and section 4 discusses the `Sip4J` permission inference technique. Section 5 explains the Pulse integration requirements with `Sip4J`. Section 5.2 elaborates on the permission inference technique using a working example and presents its evaluation using Pulse. Section 6 demonstrates the correctness and the effectiveness of the proposed technique on benchmark applications. Section 7 explains the limitations of the proposed analysis. Section 8 discusses the related work. Finally, we conclude the paper in section 9 and propose some future directions.

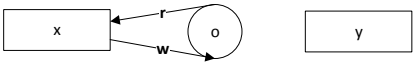
2. Preliminaries

This section briefly introduces access permissions and how different access permissions can co-exist with each other. It explains how permission-based contracts can be written following Linear Logic and the `Design by Contract principle`. Furthermore, it gives an overview of Pulse, a permission-based verification tool that is used for evaluation of the proposed technique.

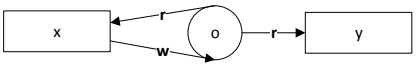
2.1. Access permissions

According to Beckman [2], access permissions can be used to describe whether or not an object is being aliased, whether a given reference can modify the referenced object, and whether other references (aliases) that point to the same object, if any, are allowed to modify the object. Let `x` and `y` be the current and other reference respectively and let `o` represent a referenced object. There are five (symbolic) permission types that can be assigned to a reference `x` for the referenced object `o` in the presence of the alias `y`.

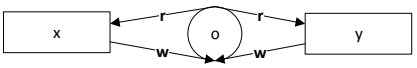
`unique(x)`: This permission provides to reference x an exclusive read and modify access on the referenced object o at any given time. No other reference (e.g. y) to the same object can co-exist while x has unique permission on o.



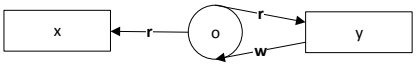
`full(x)`: This permission grants reference x with read and write access to the referenced object o, and at the same time o may also be read, but not written, by other reference y.



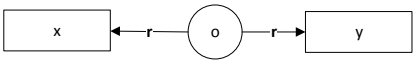
`share(x)`: This permission is the same as full(x), except that now other references y can also write on the referenced object o.



`pure(x)`: This permission gives a reference x read-only access on a referenced object o. Moreover, other reference y may have read and write access on the same object.



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



2.2. Co-existing access permissions

Table 1 below summarises how access permissions on a referenced object o can co-exist between *this reference* x and the *other references* y [1]. It shows that the unique access permission is very restrictive, as it consumes all the read and write permissions on a particular object and does not share any access with any other reference on the same object. In contrast, pure is the least restrictive as it only consumes read permission and at the same time shares read permission and gives other references exclusive right to write on the same object.

Table 1: Co-existing access permissions on the same object [1].

This reference (x)	**Access rights of (x)**	**Other references (y)**
unique	read/write	none
full	read/write	pure
share	read/write	share, pure
pure	read	full, pure, immutable
immutable	read	immutable, pure

2.3. Access permissions in the spirit of Linear Logic and the Design by Contract Principle

Access permissions are considered as resources in the spirit of Linear Logic [23]. Once a method consumes its permissions they are no longer available to other methods, until this

method produces the same permissions again. Permission-based predicates in Linear Logic are specified using implication connective (—∘). As indicated by `P —∘ Q`, permissions in the pre-conditions (`P`) are consumed when a method is called, and results in the post-conditions (`Q`) when the method completes its execution. Likewise, permission-based specifications at method level represent *contracts* in the `Design by Contract principle` [24], where 'contracts' are defined as obligations and rights of the caller and callee method. In the `Design by Contract principle`, permission-based obligations are defined as pre-conditions that the client of a class must guarantee before calling methods of that class while permission-based rights represent post-conditions that must hold for both the client and the implementing class after executing the specified method. Permission-based contracts have been specified in many formal verification approaches such as JML [25], Plural [3, 26] and Pulse [19] etc., using `'requires'` and `'ensures'` clause. The idea of specifying pre and post conditions as 'contracts' dates back to Hoare's work [27] on formal verification of software applications and recently applied to the automatic parallelization of sequential programs [8].

2.4. Plural and Pulse

In our work, we are interested in inferring permission-based specifications for Java programs. To the best of our knowledge, there are two research tools that directly support permission-based annotation for Java programs, namely Plural [4] and Pulse [19].

Plural (Permissions Let Us Reason about Aliases) [3, 26] is a formal specification language originally developed to ensure protocol compliance in typestate-based programs such as Java APIs. The program verification in Plural is based on permission-based typestate contracts where permissions define the read and write behaviour of a method on the referenced objects and typestates describe the set of valid object's states a method can be called on [28]. Plural supports five types of symbolic permissions such as unique, immutable, full, pure and share as a part of method specifications.

Pulse [19] is a permission-based verification tool that verifies correctness of Plural specifications, i.e., a Java program with permission-based contracts and typestate information in isolation to program code. It uses EVMDD-SMC model-checker [29] to ensure the integrity and the correctness of the permission-based specification by ensuring that an access permission does not violate its intended semantics. Furthermore, Pulse performs permission-based concurrency analysis of the input program, without actually analyzing the control flow dependencies between methods, at class level and compute the number of immutable methods, i.e., methods that don't produce side effects.

Figure 1 shows a workflow to verify Plural specifications through Pulse.

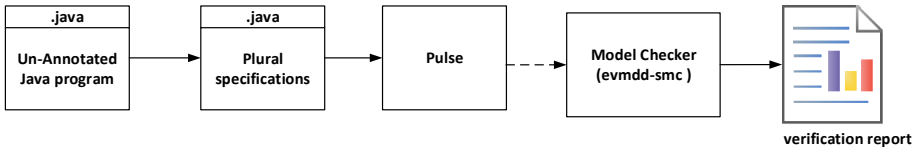


Figure 1: A high level work flow of Plural and Pulse.

Pulse takes a Plural specifications as input to verify the correctness of permission-based specifications. The permission contracts are annotated before the method declarations in the

Java program. The annotation `@Perm` is used to specify a permission contract where pre and postconditions are defined using `'requires'` and `'ensures'` keywords 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. Plural supports all five types of permissions mentioned in section 2.1. Following the `Design by Contract principle`, the precondition in a method contract specifies the type of access permissions (`AP`), a method requires on a referenced object `o` and the typestate (`s`), a referenced object should be in before the method starts its execution. The permissions (AP) on a parameter are represented using notation AP(#i) where i is an integer that maps the number of parameters as $0 \rightarrow N-1$. The postcondition specifies the permissions (`AP'`) that the method generates on the referenced object (the parameter) to return it back to the method caller and the typestate (`s'`) an objects 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 multiple classes in a program and is used by Pulse to perform correctness and concurrency analysis of permission contracts at class level.

Listing 1: A sample code snippet for Plural annotated Java methods in Pulse [21]

```
1  import edu.cmu.cs.plural.annot.*;
2  @State(name = "alive")
3  public class Task {
4   private TaskData data;
5   @Perm(ensures= "unique(this) in alive")
6   public Task(){
7    this.data = new TaskData();
8   }
9   @Perm(requires="full(this) in alive * pure(#0) in alive",
10        ensures= "full(this) in alive * pure(#0) in alive")
11  public void setData(TaskData d){
12   this.data = d;
13  }
14 }
15 ENDOFCLASS
```

Listing 1 shows a sample class `Task` taken from a case study, Multi-threaded Task Server (MTTS) [21] with Plural specifications, using single typestate '`alive`'. MTTS is a massively concurrent application, developed by Novabase to parallelise computational tasks and verified recently by Siminiceanu et al., [19] using Pulse. In MTTS, the class `Task` captures all the information about a generic task in a data structure `data` (line 4). The permission contract (line 5) for the constructor method `Task()` specifies that when the constructor is called it does not require any permission on the current object `this`. There is no `'requires'` clause but it `ensures` to generate a unique permission, represented by unique(this), on the current object (`this`) in the 'alive' state. Likewise, the precondition of method `setData()` (line 9) specifies that the method should have full permission on the referenced object (`this`) in the 'alive' state. It also `requires` that the method needs pure permission on parameter '`d`' represented by pure(#0), as the method does not change the state of this parameter object. The postcondition (line 10) specifies that the method generates the same permissions (full(this) & pure(#0)) on the referenced objects to return the consumed permissions back to the method caller.

To verify the correctness of the permission-based specifications and to perform the concurrency analysis, Pulse considers the following:

1. **Satisfiable Methods** A method is satisfied if all its preconditions in the `requires` clause are met. Its satisfiability analysis is performed using the `requires` clause to check whether the method can be reached from another method based on this permission contract.
2. **Unsatisfiable Methods** The presence of an unreachable or unsatisfiable method indicates that no possible client can fulfil the method's contract i.e., the `requires` clause and this method is not called under any circumstances; thus the method remains unreachable. The presence of the unreachable method is due to the method's unsatisfiable pre-conditions which indicates an error in the inferred specifications.
3. **Concurrent Methods** Based on the permission constracts and following permission co-existence rules (Section 2.2), Pulse computes the superset of immutable methods which are methods that never change the state of the shared objects or do not not produce side effects. It therefore computes the possible concurrency among methods.

3. The `Sip4J` Framework

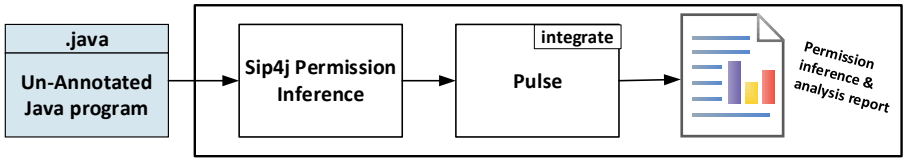


Figure 2: A high-level work-flow depiction of the Sip4J framework

We propose the `Sip4J` framework which extends Pulse with a permission inference engine, as shown in Figure 2. The engine performs modular static analysis of the Java source code based on Abstract Syntax Tree (AST) to automatically reveals implicit dependencies present between the code (methods) and the global states and maps them in the form of permission-based dependencies (contracts) using graph abstractions. By integrating with Pulse, `Sip4J` evaluates the correctness of the inferred specifications. Pulse takes the annotated Java program with Plural specifications (permission-based contracts). It uses EVMDD-SMC model-checker to generate abstract state machine model of the given specifications and performs reachability graph analysis of the generated state space to verify the correctness of the input specifications. As such, Pulse analyses the concurrent behavior of the sequential programs based on permission-based specifications.

Our permission inference technique and its integration with Pulse is explained in more detail in Section 4 and Section 5 respectively. To this end, the `Sip4J` framework produces the following artifacts from an un-annotated Java program:

- Five types of symbolic permissions at the class field level, for all the referenced variables accessed inside a method from its global environment, considering everything as an object.
- A Pulse translated (Plural annotated) version of the input Java program where permissions are generated at object level.
- A verification report that comprises the correctness and the concurrency analysis of the inferred specifications, by Pulse, along with Plural annotated version of the input program.

4. The `Sip4J` permission inference technique

To generate access permissions for the global states at methods level, following permission semantics (Section 2.1), we need to identify the way (`read` or `write`) a referenced variable `o`, is accessed by the current method e.g., `x` and, at the same time, by its context (global environment) e.g., `y`. Moreover, we need to identify and track aliases of the reference(s) (if any) to maintain integrity of the data and to identify correct dependencies at method level. The extracted information is then modeled in a graph structure to generate access permissions, as *pre* and *post* permissions for the global references accessed in a method.

Figure 3 shows a high-level architecture of the `Sip4J` permission inference engine, which accepts the un-annotated Java program as input and produces the annotated Java program with permission-based specifications as output. For this purpose, the proposed technique includes the following sequential steps or phases as shown in Figure 3:

Metadata Extraction. It parses the AST of the Java source code to extract and maintain the metadata information such as data flow, alias flow and context information, for all the global references accessed in a method (Section 4.1).

Graph Construction. For each method, based on the extracted information, it constructs a directed graph, using graph-based formal notations and by following syntactic rules for graph construction (cf. Section 4.2).

Graph Traversal. It traverses the constructed graph for each method and generates symbolic permissions for the global referenced variables accessed at method level using permission inference rules (cf. Section 4.3).

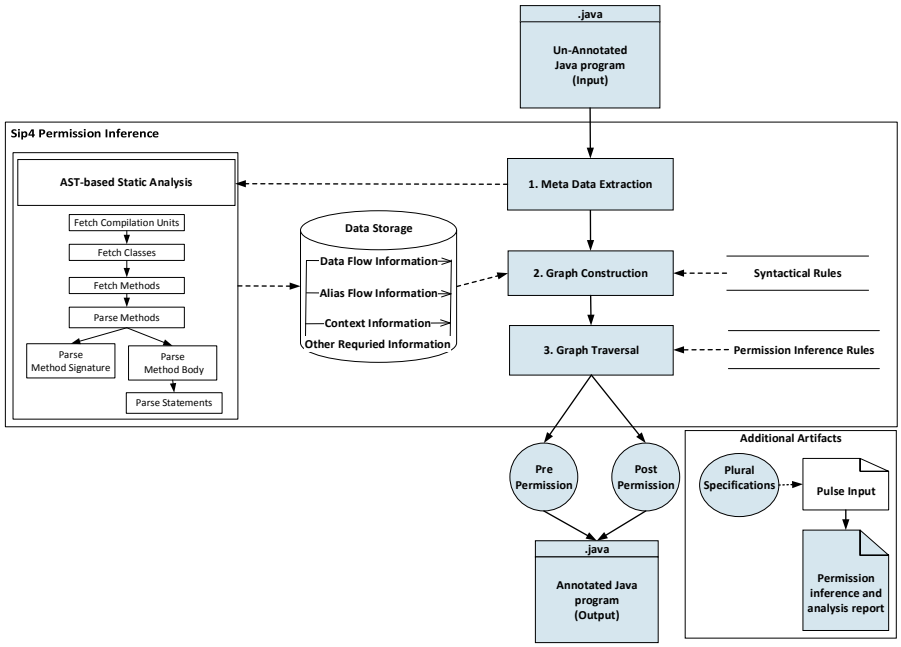


Figure 3: The `Sip4J` permission inference engine.

In the following sections, we will elaborate on each of the three stages of the above mentioned permission inference technique using a running example shown in Listing 2.

Listing 2 shows a sample Java code snippet with two classes `A` and `B`. Class `B` has an attribute `data` of type `Integer`. Class `A` composes class `B` through references `x`, `y`, `z` and `w` at line 5. Class `A` has a method `foo()` that manipulates its data members and `main()` method acts as client method for method `foo()`. We will apply our technique on method `foo()`, assuming that all the references accessed in it are non-null references.

Listing 2: A code snippet with method `foo()` and its client method `main()`.

```
1 class B {
2   Integer data = 100;
3 }
4 public class A{
5  B x,y,z,w;
6  B foo(B p1, B p2){
7   x = p1;
8   B t = x;
9   y = t;
10  x.data = 10;
11  System.out.println("z.data = "+p2.data+"x.data = "+t.data);
12  return y;}
13  public static void main(String[] args){
14  A obj = new A();
15  obj.foo(obj.w,obj.z);}
16 }
```

4.1. Metadata Extraction

We perform data flow and alias flow analysis of the source code to extract the metadata information (read, write and alias information) for all the global references accessed at method level in `Sip4J`. The analysis is based on the type of expression such as `<FIELD_ACCESS>`, `<ASSIGNMENT>` etc., encountered in an expression statement and the type of referenced variable(s) accessed in each expression, such as a global reference variable (`<grv>`) i.e. a class field or a method's local references (`<lrv>`) that is an alias of a global reference as manipulating a local reference that does not refer to a global object, does not affect the access rights of other methods (see graph conventions Section 4.2.1).

The context analysis of a referenced variable accessed in method is based on its data flow and alias flow analysis across all other methods. There can be three possible contexts for a referenced variable, represented as `Context-N` (no access), `Context-R` (read access) or `Context-RW` (read and write access) in `Sip4J` that define its access by other methods. The type of permissions generated in each context (`Context-N`, `Context-R` or `Context-RW`) depends on whether current method say `foo` access (modifies) the global references say `<grv>` or `<lrv>` etc.

Starting with an input Java program, the technique parses the AST of the program to fetch the set of compilation units (*CU*) with a set of user-defined classes (*C*) and set of methods (*M*) in each class. Each method (*m*) is then parsed with its signature and body to fetch and store the data flow, alias flow and context information for all the referenced variables accessed in a method from its global environment. Different versions of graph construction rules are then applied such as `(GR-Read-Only,<grv>)`[2], `(LR-Add-Flow,<grv>)`[3] etc.,

[2]GR rules captures expressions having global references or class fields.

[3]LR rules capture expressions having local references that are alias of global references.

to map the extracted information in a graph model, based on the type of expression and the type of referenced variable encountered in each expression. The details of the parsing of method signature and the parsing of method body are described below:

Method Signature: Here, the method name, the return type, the visibility modifier and formal parameters are all stored in a data structure. While parsing the method signature, we map the formal parameters with their corresponding arguments by fetching method invocations of the same method at project level. For each parameter and for each invocation of the method, we verify the type (`<grv>`, `<lrv>`) of argument to be a global reference or alias of some global reference. This information is used to fetch (maintain) the data flow and alias flow information of the actual object against parameter and to ensure the integrity of data during parsing while the same object is accessed by other methods in the program. For example, in Listing 2, the proposed technique first maps the formal parameters `p1` and `p2` with their actual objects i.e. `w` and `z` respectively. As `w` and `z` are global references `<grv>`, their access is maintained as a read access by the current method `foo`. This information is then used to create a graph structure for method `foo`, following `(GR-Read-Only, <grv>)` for both objects during graph construction phase (see Figure 4).

Method Body: In parsing a method body, the technique recursively parses each statement (*s*) in a method (*m*) body. Each expression in an expression statement is recursively parsed to distinguish between the `<read-only>` and `<read-write>` expression and parse it accordingly. The handling of read and write expressions in a statement are described:

- The `<read-only>` expressions are characterized by `<FIELD_ACCESS>` and `<QUALIFIED_NAME>`, `<SIMPLE_NAME>` etc., expression nodes in AST. The proposed technique recursively parses an expression statement to fetch the read-only expressions and the referenced variables in it say `<grv>` or `<lrv>`. This information (read access) is maintained in a data structure as a part of data flow analysis at method level.
  - For example, in Listing 2 at line 11, for expression `p2.data` and `t.data` the proposed technique first maps the parameter `p2` and method's local reference `t` with their global references i.e., `z` and `x` respectively. This information is maintained as read access by the current method (`foo`) for `z` and `x`. Similarly , for return statement (`return y;`) at line 12, we store read access for variable `y`, being the global reference, by current method `foo`. The extracted behavior is then modeled in a graph structure for method `foo` following the `(GR-Read-Only,<grv>)` for object `z` and `y` and `(LR-Read-Only,<lrv>)` rule for object `x`.
- The `<read-write>` expressions are characterized by `<ASSIGNMENT>` expression in AST. The proposed technique performs flow-sensitive analysis of the source code, where type of a reference on the left hand side of an assignment statement is determined based on its right hand side expression. The proposed technique recursively parses the right and the left hand side of an assignment expression to fetch the read, write and alias information for all the reference variables accessed in each expression. During parsing, we categorize the assignment statement as a `<value-flow>`, `<object-creation>`, `<address-flow>`, `<null-address-flow>`, `<self-address-flow>` etc., statement based on type and data type of the right hand side expression (operand).
  - For example, in Listing 2 at line 7, we categorise the expression `x = p1;` as <address-flow> expression with address flowing to global reference `x`. This is because the right hand side of this expression is a reference type. As mentioned earlier, we map each formal parameter with its actual global reference. As object `p1` is an alias of `w` which

means `x` now points to `w` through `p1`. This information is maintained in `Sip4J` as part of the alias flow analysis for object `w`, as any change in the state of object `w`, made directly or indirectly through `p1` can affect reference `x` and its aliases. Further, the access for the referenced variable `w` is stored as read access by the current method `foo`. Such expressions are model by following (GR-Addr-Flow, `<grv>`) address flow rule during graph construction.

- Similarly, at line 8, the expression `B t = x;` is categorized as <address-flow> statement for the local reference `t` that now refers to a global reference `x`. We maintain and track this information as a part alias-flow analysis to ensure the integrity of data during parsing. The analysis maintains a pointer-pointee relationship between `t` to `x` as any change in the state of reference `t` can affect the state object against reference `x`. The access for the right hand side operand i.e., `x` is handled as a `<read-only>` expression. This information is modeled as a part of graph structure for method `foo()` by following (LR-Addr-Flow, `<grv>`) rule during graph construction phase.
- For expression `y = t;` at line 9, the proposed technique maps the local reference `t` with its global reference i.e., `x` which means that reference `y` is now an alias of reference `x`. This information is maintained as a part of alias flow analysis to keep track of the changes made by reference `y` that can eventually affect reference `x` through `t` and reference `w` through `x`. The expression at line 8 is modeled in a graph using `(GR-Addr-Flow,<lrv>)` rule.
- Further, at line 10, the expression `x.data = 10;` is treated as a <value-flow> statement as right hand side of this expression is a <Literal> constant. The proposed technique maps it as a write access on reference `x` by the current method. Further, the analysis ensures that this change (write operation) should be propagated to all the aliases of reference `x` to map (update) their access accordingly. When the right hand side of an assignment statement yields a `<PrimitiveType>` or `<Literal>`, the technique models the write access of reference `<grv>` in a method graph by following `(GR-Val-Flow,<grv>)` rule.

- The method calls are handled using expressions such as `<METHOD_INVOCATION>` and `<SUPER_METHOD_INVOCATION>` etc., in AST. As a part of modular analysis, the permission inference technique is recursively applied to every method invocation *sub_m* (a non-recursive method call) in caller method. The technique saves the current state of caller method in a data structure. The analysis of the caller method is completed when all of its sub-methods have been parsed and the permission on their referenced variables have been generated. For sub-method calls, we follow the appropriate method call rules such as `MethodCall(<Full>, <grv>)` (see Section 4.2) to model graph of the caller method. Other expressions are handled similarly by applying appropriate rules.

It is worth mentioning here that we skip recursive method calls in a method statement. This is because a recursive method call does not change the way a method accesses the reference variables in its body, reducing the analysis time as well. The super method calls, in the case of inheritance, are handled the same way as other non-recursive method calls but at the moment, our analysis does not handle overridden method calls (when child overrides a method in the parent), and hence, dynamic method dispatch[4]. However, static method

[4]In Java, dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at

dispatching to multiple targets is straightforward and is determined using method signature and the static type of the referenced object at compile time. We envision that the analysis can be extended to decide which overridden method should be parsed as a result of overridden method call, and this is because we parse the type of expression on the right hand side of an assignment statement during analysis.

4.2. Graph Construction

The graph construction phase maps the extracted data flow, alias flow and context information in the form of a graph model. For this purpose, we define graph based abstractions and the graph construction rules which are described below. Figure 4 shows a graph model generated by `Sip4J` for method `foo` given in Listing 2, during graph construction phase.

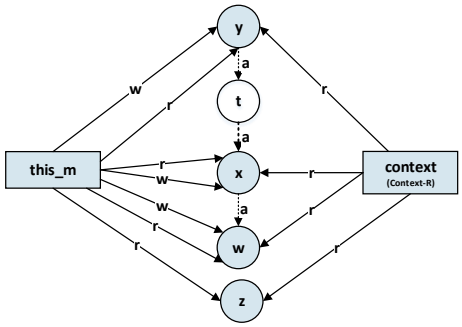


Figure 4: The graph model of method foo() in `Context-R`

4.2.1. Graph notations, conventions and concepts

For graph construction, we use some special nodes, edges and concepts to construct graph model which are described below:

A. Method Nodes: We use two types of nodes to represent the current method `x` and its global environment `y` as follows.

1. `this_m` is an abstraction that is used to represent the current method being parsed. It is represented as a labelled rectangle. For example, method `foo` in Listing 2 is labelled as `this_m` in Figure 4.
2. `context` is an abstraction, depicted as a labelled rectangle, that represents a method's global environment accessing the same reference objects. For example, we choose read context (`Context-R`) for all the referenced variables accessed in method `foo` to elaborate its permission inference mechanism.

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

B. Variable Node: It models a reference variable accessed by a method. It is depicted as a labelled circle with variable name (ID). A reference variable node can be one of the following three types:

---

runtime and the version of the overridden method to execute is based on the type of object it refer to

1. <grv> is a reference variable accessed by a method from its global environment. It can be a class field or a parameter that is an alias of a class field. For example, x and y are class fields in Figure 4. The reference variable can be a primitive type or a reference type.
2. <lrv> is a local reference variable declared inside a method but that is an alias of a global reference. For example, variable t in Figure 4 is an alias of global reference x.
3. <lv> is a local variable declared in a method other than <lrv>. In Listing 2, no <lv> is declared in method foo.

C. Edges: There are two types of edges:

1. **read/write edges** model the read or write access on the global referenced variables either by the current method this_m or by its context or both. They are depicted as directed and solid edges labelled as 'r' or 'w'. For example, a read edge and a write edge from this_m and variable node x in Figure 4. These edges are being referred as ReadEdge(this_m, <grv>) or WriteEdge(this_m, <grv>) in the graph construction rules.
2. **alias edge** models an alias[5] of a references, if any. The alias edge is depicted as a directed and dotted edge labelled with letter 'a' and depicted as AliasEdge(<grv>, <grv1>). For example, x is an alias of w in Figure 4. The convention AliasEdge(<grv>, <grv1>) also shows a pointer-pointee relationship between two references.

4.2.2. Sip4J rules for graph construction

In Sip4J, we define syntactic rules to generate a graph model for each method from the metadata extracted in the first stage of permission inference technique. These rules are categorized into two types depending on their usage: a) *Context* rules, b) *Statement* rules. The statement rules, for simplicity, are further categorized as *method* call and *non-method* call statements Further, the statement rules for graph construction are based on the type of expression statement and the type of reference variables accessed in it. Although the rules are self explanatory, we describe some to provide an intuition on specifying the underlying process for graph construction.

A. The **Context** rules specify how to add read and write edges between context and variable nodes (<grv>), following the variable access by other methods. For example, the (Context-R, <grv>) rule specifies that we need to add a read edge from context to <grv> node to show that the global reference variable is being read by its context.

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

B. The **Statement** rules are designed to follows the style of sequent calculus, as shown below.

$$\frac{\texttt{<Exp-Statement>}}{\texttt{<Rule-Description>}}\ \texttt{(<Rule-Name>, <grv>)} \qquad (1)$$

These rules describe different ways to add edges according to the type of statement <Exp-Statement> encountered and the way <Rule-Description> a referenced ob-

[5] An alias can be another reference (<grv1>), the current reference itself (<grv>) or a local variable (<lrv>) in a method.

ject <grv> is accessed (read and write) by the current method and its context, if any. The rule's name (<Rule-Name>, <grv>), itself follows the type of reference variable (GR for <grv>, LR for <lrv> and L for <lv>) accessed in an expression. Some of the statement rules are described below:

- The **(GR-Read-Only,** <grv>**)** rule models the read access on the reference variable (<grv>) by the current method this_m.

$$\frac{\texttt{<grv>}}{\texttt{addReadEdge(this\_m,<grv>)}}\ (\textbf{GR-Read-Only,}\texttt{<grv>})$$

- The **(GR-Val-Flow,** <grv>**)** rule models write access on reference <grv> by the current method (this_m). This rule states that we should add a write edge from this_m to the <grv> node. It further ensures that this change should be propagated to all the aliases of <grv> to maintain data integrity. Therefore, in the graph, we need to add a write edge from this_m node to all its alias nodes, if any. The referenced variables on right hand side of assignment statement are modeled as read-only expressions.

$$\frac{\texttt{[Type] <grv> = <Literal>|<grv1> |MethodCall(<post-perm>,<grv1>)}}{\begin{array}{l}\texttt{addWriteEdge(this\_m,<grv>)}(\forall a \in \texttt{aliasOf(<grv>)addWriteEdge(this\_m, a)}),\\ \texttt{(apply(GR-Read-Only,<grv1>)|apply(MethodCall(<post-perm>,<grv1>)))}\end{array}}\ (\textbf{GR-Val-Flow,}\texttt{<grv>})$$

- The **(GR-Addr-Flow,** <grv>**)** rule models expression of the form <grv> = <grv1>. The rule states that we should add an alias edge from <grv> to <grv1> and should remove any existing alias edge from <grv> to <grv2>. This information is maintained as a part of alias-flow analysis during parsing.

$$\frac{\texttt{<grv> = <grv1>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<grv>,<grv2>)} \Longrightarrow \texttt{removeAliasEdge(<grv>,<grv2>))}\\ \texttt{addAliasEdge(<grv>,<grv1>),apply(GR-Read-Only,<grv1>)}\end{array}}\ (\textbf{GR-Add-Flow,}\ \texttt{<grv>})$$

- The **(LR-Addr-Flow,** <grv>**)** rule applies to an expression of the form <lrv> = <grv>. The rule states that we should add an alias edge from local reference <lrv> to global reference <grv> and should remove existing alias edge, if any, from <lrv> to <grv1>. The proposed analysis keeps track of changes in <lrv>, as a part of alias-flow analysis, that could affect the global reference and its aliases.

$$\frac{\texttt{[<Type>] <lrv> = <grv>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv1>)} \Longrightarrow \texttt{removeAliasEdge(<lrv>,<grv1>)),}\\ \texttt{addAliasEdge(<lrv>,<grv>),apply(GR-Read-Only, <grv>)}\end{array}}\ (\textbf{LR-Addr-Flow,}\ \texttt{<grv>})$$

- The **(GR-Addr-Flow,** <lrv>**)** rule is applied to expressions of the form <grv> = <lrv> where <lrv> is an alias of a global reference, say <grv1>. It states that we need to add an alias edge from <grv> to <lrv> and should remove the existing alias edge, if any, from <grv> to any other global reference, say <grv2>.

$$\frac{\texttt{[<Type>] <grv> = <lrv>}}{\begin{array}{l}\texttt{(}\exists\texttt{aliasEdge(<grv>,<grv2>)} \Longrightarrow \texttt{removeAliasEdge(<grv>,<grv2>)),}\\ \exists\texttt{aliasEdge(<lrv>,<grv1>)} \Longrightarrow \texttt{addAliasEdge(<grv>,<lrv>)),apply(GR-Read-Only,<grv1>)}\end{array}}\ (\textbf{GR-Addr-Flow,}\ \texttt{<lrv>})$$

C. The **Method Call** rules specify how to add read and write edges in a caller method as a result of a method call expression in the method statement. The type of edges added in the called method graph depends on the post access permissions (<post-perm>) of the reference variables (<grv>) accessed in a called method. For example, the Method-Call(<Full>, <grv>) rule generates full permission on its reference objects (<grv>) as post permissions. To add this method in the caller graph, following the semantics of full permission, we need to add both read and write edges from the called method (this_m) to <grv> node and a read edge from context to <grv> node following (Context-R,<grv>) rule for <grv>.

$$\frac{\texttt{MethodCall(<grv>)}}{\texttt{addWriteEdge(this\_m,<grv>),addReadEdge(this\_m,<grv>),apply(Context-R,<grv>)}}\ \textbf{MethodCall(Full,<grv>)}$$

4.3. Graph Traversal & Permission Generation

The Access **permission inference** rules are used to generate five kinds of symbolic permissions. This is done by by traversing the graph of each method where the type of access permission generated on a reference variable depends on the type of edges between the current method and the variable node and the presence (or absence) of alias edges between variable nodes. For example, the **(Full, <grv>)** permission inference rule states that:

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

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

Listing 3 shows the access permissions generated by Sip4J for method foo by traversing its graph model (Figure 4).

Listing 3: Permission contract for method foo() in Pulse format

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

Other kinds of access permissions are generated using their corresponding rules. The complete list of graph construction and permission inference rules is attached in Appendix B.

4.4. Termination Analysis

We highlight some behavioural properties in regards to the termination of analysis:

- The proposed analysis always converges to an expression type designated as a base case such as <FIELD_ACCESS>, <SIMPLE_NAME> or <ASSIGNMENT> etc., while parsing expressions in AST.
- We skip method invocation expressions for recursive calls as mentioned previously. In the case of infinite and chained recursive calls (eg. when a method foo1() calls method foo2() which again calls foo1()), the analysis terminates successfully. This is because, during parsing we maintain the method's metadata which helps identify and skip the second level, i.e., indirect recursive call for method foo1() through method foo2() and continue parsing from the next expression in the caller method foo2().

- The proposed technique successfully terminates the metadata analysis when required and eventually completes the graph construction process. For example, during analysis when a reference (direct or indirect) starts pointing to itself, or to a local reference that is not an alias of a global reference or when a reference becomes a null reference, the proposed technique will follow the appropriate syntactic rules such as `(GR-SelfAddr-Flow, <grv>)`[6] and `(GR-NullAddr-Flow, <grv>)`[7], to identify loops, cycles and null references in the underlying program and eventually terminates the graph construction process.
- Graph construction rules are developed in such a way that they guarantee no cycle or loop in the constructed graph. Therefore, they make graph traversal simple and there is no backtracking in the constructed graph to ensures the termination of analysis and the inference mechanism itself.

5. Integration of Pulse in `Sip4J`

As discussed previously in Section 2.4, we can check the correctness of inferred specifications and the potential for concurrency using Pulse. For this purpose, we integrate Pulse as a part of the `Sip4J` framework. As Pulse takes Plural annotated Java programs as input, we automatically generate Plural specifications of a given Java program from permissions inferred by `Sip4J`. Section 5.1 explains in detail the Plural translation of a Java program. Section 5.2 illustrates the translation process with a running example. This is followed by its analysis in Section 5.3.

5.1. Pulse Translation

A number of characteristics of Pulse and Plural are worth noting.

Firstly, Plural is a permission-based specification language where access permissions are defined at object level using keyword "`this`". It does not support permissions defined at the field level e.g. `pure(this.field)` and `pure(field)` which can be the case in Java programs. This is different from our `Sip4J` framework, which generates permissions at the field level, considering everything as an object, and thus can enable concurrency at a more granular level in the future.

Secondly, Pulse tool does not support permissions share and immutable. This is because, referring to the permission coexistence semantics in Table 1, share cannot coexist with either unique, full or immutable, and immutable cannot coexist with either share or full. Hence, Pulse's concurrency analysis of methods cannot be performed on these annotations.

Thirdly, Pulse follows the `Design by Contract principle` and support permission contract of the form `P` $\Longrightarrow$ `Q` where the relation $P \stackrel{!}{=} Q$ (the pre and post permissions should be same to fulfill the contract between caller and callee method) must hold to maintain integrity of the specifications during analysis but the situation can be different when permissions are actually inferred from a source program. For example, in Listing 5 line 15, `Sip4J` generates permission contract for method `tidyupColl()` with different pre and post permission as there is a null address flow statement for reference `coll`. In this case, we generate 'none'

[6] When expression statement is of the form `<grv> = <grv>;`

[7] when expression statement is of the form `<grv> = null;`

permission as post permission to show that `coll` is a null reference now and a new instance of the array object needs to created with unique permission on reference `coll` to used it again in the program.

To reconcile the differences between `Sip4J` and Pulse, `Sip4J` automatically performs the following tasks to integrate Pulse as part of our framework.

- It uses permissions inferred at the fields level to generate a conservative and safe permission for the current instance of object i.e. `this`. For this purpose, we compute the maximum of the pre and post permissions of all the referenced variables (class fields) associated with the current instance of object (`this`).
- It generates notation `<AP>(#i)` to represent <permission> on parameter fields with respect to their position. For example, the notation pure(#0) in Listing 6 line 11 shows permission specified for the first parameter.
- It converts share permissions, generated in the case of `Context-RW`, for reference variables to full; and immutable permissions, generated in the case of `Context-R`, to pure.
- It converts inferred contracts following the relation $P \stackrel{!}{=} Q$ where pre (P) and post (Q) permissions are the same (Listing 6 line 17).
- It defines a typestate '`alive`', which is the default global (root) typestate of an object, using `@States` statement at the class level (Listing 6, line 3).
- It generates other required annotations including:
  - Typestate as a part of method contract (Listing 6, line 5 and 7)
  - The annotation `@ENDOFCLASS` at the end of every class (Listing 6, line 19).
  - Import statements to import the required packages to support Plural annotations (Listing 6, line 2).

The example in Listing 6 also shows the minimum annotation overhead imposed by the existing permission-based verification approaches in Plural and Pulse.

5.2. An Example

We illustrate the Pulse translation and analysis mechanism using an unannotated Java program with a user defined array collection class `ArrayCollection` class, shown in Listing 4. For completeness, the client classes for both annotated and unannotated versions of this program are included in Appendix as Listing 7, 8 and 9.

The `ArrayCollection` class has a data member `coll` of type `Integer` array and seven methods manipulating the `coll` object. Listing 5 shows the annotated version of the same class with permissions generated by the `Sip4J` framework. In Listing 5, the inferred permission contracts are given as comments, using linear logic implication `P` $\multimap$ `Q` to represent the pre (`P`) and post permissions (`Q`) on each individual class fields including parameters accessed in a method.

Consider the permission contract "`pure(coll)`$\multimap$`pure(coll)`" associated with the method `displayColl()` (Listing 5, line 7). It states that the method needs pure as pre-permission on `coll` and it generates the same permission (pure) as the post-permission when it exits.

Listing 4: The un-annotated ArrayCollection class in Java.

```
1 package outputs;
2 public class ArrayCollection{
3    public Integer[] coll;
4    public ArrayCollection(int size){
```

```
5         coll = new Integer[size];}
6     public void initColl(){
7        for(int i = 0; i < coll.length; i++)
8          coll[i] = (int)(Math.random()*10);}
9     public void displayColl(){
10       for(int i = 0; i < coll.length; i++)
11          System.out.println("coll[i] ="+coll[i]);}
12    public void displayE(Integer[] e) {
13       System.out.println(""+e[0]);
14    }
15    public  void initE(Integer[] e) {
16       for(int i =0; i < coll.length; i++)
17         e[i] = coll[i]*2;}
18    public void copyColl() {
19       Integer[] temp = coll;
20       for( int i = 0 ;i < temp.length; i++)
21       temp[i] = temp[i]+i;}
22    public void tidyupColl() {
23       coll = null;}
24 }
```

Listing 5: The field-level permission annotated version of the ArrayCollection class given in Listing 4.

```
1  package outputs;
2  class ArrayCollection {
3     //none(coll)—⊸unique(coll)
4     ArrayCollection(){... }
5     //share(coll)—⊸share(coll)
6     public void initColl(){ ... }
7     //pure(coll)—⊸pure(coll)
8     public void displayColl(){ ... }
9     //pure(e1)—⊸pure(e1)
10    public void displayE(Integer[] e){ ... }
11    //pure(coll)*full(e1)—⊸pure(coll)*full(e1)
12    public void initE(Integer[] e1){ ... }
13    //share(coll)—⊸share(coll)
14    public void copyColl(){ ... }
15    //unique(coll)—⊸none(coll)
16    public void tidyupColl(){ ... }
17 }
```

5.3. *Pulse Analysis of the Example*

In this section, we describe the analysis Pulse performed on the Plural-translated version of the `ArrayCollection` class, shown in Listing 6. The results of the analysis is shown in Figure 5.

Listing 6: The Pulse translated version of `ArrayCollection` class given in Listing 5.

```
1  package outputs;
2  import edu.cmu.cs.plural.annot.*;
3  @States({@State(name = "alive")})
4  class ArrayCollection {
5   @Perm(ensures="unique(this) in alive")
6   ArrayCollection(){}
7   @Perm(requires="full(this) in alive", ensures="full(this) in alive")
8   public void initColl() {}
9   @Perm(requires="pure(this) in alive",ensures="pure(this) in alive")
10  public void displayColl(Integer[] coll){}
11  @Perm(requires="pure(#0) in alive", ensures="pure(#0) in alive")
12  public void displayE(Integer[] e){}
13  @Perm(requires="pure
        (this)*full(#0) in alive",ensures="pure(this)*full(#0) in alive")
14  public void initE(Integer[] e1){}
15  @Perm(requires="full(this) in alive", ensures="full(this) in alive")
16  public void copyColl(){}
17  @Perm(requires="unique(this) in alive",ensures="unique(this) in alive")
18  public void tidyupColl() {}
19 }ENDOFCLASS
```

5.3.1. Correctness Analysis

Pulse verifies the correctness of permission-based specifications by performing the satisfiability analysis for each method based on permission contracts.

Method satisfiability analysis: Figure 5a shows the method satisfiability analysis results for the example program given in Listing 6. It shows that all methods are satisfied with their permission contracts, which means that every method has its required (pre-) permissions generated by some other method, which in turn ensures the absence of unreachable method in the program.If a method is unsatisfiable, either (a) its pre-condition (`requires` clause) itself is wrong, or (b) the post-condition (`ensures` clause) of some other method accessing the same reference is wrong or missing.

For example, in line 9 of Listing 6 line 9, it can be seen that method `displayColl()` needs pure as pre-permission for the reference object `this`. The method would become unsatisfiable if no other method generates pure, full or unique on the same object. This is because unique and full are more restrictive than pure, and hence they can be used to satisfy the pre-permission of method `displayColl()`. Pulse automatically identifies such errors (if any) in the specification and reports them as unsatisfiable conditions.

Null pointer analysis: The inferred specifications can help a programmer identify some of the syntactic errors in a program at compile time such as *null pointer* references. With respect to permission semantics, this situation can arise in because of: (a) `Sip4J` error: that the permission analysis itself fails to generate unique permission on the reference object, or (b) program error: that no method (including constructors) in the program instantiates the class thereby, generating unique permission on the reference object. For example, in Listing 7, if the client program does not instantiate `ArrayCollection` class before using it, the proposed technique would not generate unique permission on the referenced object `this`. This is because the underlying reference is a null reference that in turn causes all the dependent methods such as `initColl()` and `tidyupColl()`, to remain unsatisfied or unreachable as their pre-conditions would never be met.

5.3.2. Concurrency Analysis

The inferred permissions can be used to analyze the parallelizability of a Java program. For this purpose, Pulse computes a superset of immutable methods, i.e., methods that do not change the state of a shared object. Figure 5b shows the permission-based concurrency analysis matrix of the `ArrayCollection` class given in Listing 6. In the concurrency matrix, possible parallel execution of two methods is depicted by the symbol ||, whereas the symbol ∦ represents the fact that two methods cannot be parallelized with each other.

The results in Figure 5b show that six methods in `ArrayCollection` class can be parallelized with at least one other method in the same class. The constructor cannot be parallelized with any method as no other method can use the object before the object is created. Similarly, methods that require full or share access on the same object e.g. `initColl()` and `copyColl()`, cannot be executed in parallel due to the side effects they produce. On the other hand, based on permission contracts these methods can be parallelized either with the (a) methods that require only read (pure) access on the same object such as method `displayColl()` etc., or (b) methods that require (full) permission on some other object. For example, method `copyColl()` can be parallelised with `initE()` both requiring full permission on different objects. It also shows that method that requires unique permission on

a shared object cannot be parallelised with any other method. For example, `tidyupColl()` can only be parallelised with `displayE()` as both access different objects.

The concurrency analysis also reveals the fact that 13 out of 28 pairs of methods (46%) in the `ArrayCollection` class can be parallelised with each other. The permission-based concurrency analysis demonstrates the effectiveness of the inferred specifications to enable implicit concurrency from a sequential program.

Method	Satisfiability
ArrayCollection	√
initColl	√
displayColl	√
displayE	√
initE	√
copyColl	√
tidyupColl	√

(a) Satisfiability of methods.

	ArrayCollection	initColl	displayColl	displayE	initE	copyColl	tidyupColl
ArrayCollection	∦	∦	∦	∦	∦	∦	∦
initColl	∦	∦	∥	∥	∥	∦	∦
displayColl	∦	∥	∥	∥	∥	∥	∦
displayE	∦	∥	∥	∥	∥	∥	∥
initE	∦	∥	∥	∥	∥	∥	∦
copyColl	∦	∦	∥	∥	∥	∦	∦
tidyupColl	∦	∦	∦	∥	∦	∦	∦

(b) Method concurrency matrix.

Figure 5: Results of (a)Correctness analysis and (b)concurrency analysis of methods in the `ArrayCollection` class as produced by Pulse.

6. Evaluation

We have implemented the `Sip4J` framework along with its integration with Pulse as an Eclipse Plugin. We have performed an empirical evaluation of `Sip4J` by applying the framework to computationally intensive Java applications of three widely used benchmarks suites, Java Grande Benchmark (jomp) [30], Æminium[8] and Plaid[9], together with Pulse[10].

In total, `Sip4J` generated 10,791 permission annotations at field level, considering everything an object, for 999 methods in 149 classes for 20 benchmark programs and 1,094 annotated lines (permission contracts) were generated for the Pulse translated version of the same programs, where permissions are basically assigned at object level, with 3,598 annotations for 999 methods. The details of the number of annotations generated by `Sip4J` are given in Section 6.2.1.

We conducted three experiments on the benchmark suites to evaluate our framework:

1. Verify the **correctness** of the inferred specifications using Pulse (Section 6.1).
2. Demonstrate the **effectiveness** of `Sip4J` (Section 6.2), where we

[8] `https://github.com/AEminium/AeminiumBenchmarks/tree/master/src/aeminium/runtime/benchmarks/`.

[9] `https://github.com/plaidgroup/plaid-lang`.

[10] `http://poporo.javerianacali.edu.co/aeminium/pulsepulse/pulse.php`.

- measure execution time of the analysis performed by `Sip4J`, and
- calculate the amount of inferred annotations to measure annotation overhead.
- perform quality analysis of the inferred annotations by comparing them with the annotations manually generated.

3. Compare the **performance** of inferred permissions with their manually annotated counterparts for some of the benchmarks using Plaid (Section 6.3).

Table 2 shows the results of the experiments. For every program (column **P**) in the benchmark suit (column **B**), we present the results of the correctness and concurrency analysis of the inferred specifications by Pulse (columns **SM**, **USM**, **CM(%)** and **CMP(%)**) and the results of the effectiveness analysis of our technique in the last five columns (**LOA**$_P$—**ES** (sec)).

6.1. Correctness and concurrency analysis of the inferred specifications

Pulse verifies the correctness of permission-based specifications in terms of method satisfiability analysis and computes a superset of immutable methods and the methods that should run sequentially based on permission contracts.

Correctness analysis. The column **SM** shows the number of methods determined by Pulse to be satisfiable, whereas column **USM** shows the number of methods Pulse determines to be unsatisfiable. In other words, **M** = **SM** + **USM** for each program. As can be seen in the table, column **SM** confirms that `Sip4J` successfully infers satisfiable (required) specifications, without any specification errors, for all methods **M** in **C** classes with three exceptions: Pulse, montecarlo and blacksholes.

For Pulse, 18 of the 434 methods have been determined to be unsatisfiable. Upon further analysis of the Pulse source code, we noticed that the unsatisfiability is due to the fact that Pulse does not support overloaded methods as part of its analysis, even though the same methods are provided with different permission contracts, and that all these 18 methods are overloaded methods. For montecarlo, the analysis shows unsatisfiability for 17 of the 168 methods and for blacksholes the count is 2 out of 34. Again, this is due to the presence of overloaded methods. As Pulse does not support overloaded methods, therefore, we manually analyzed all the overloaded methods in the given prorgams with their inferred specifications and found them to be satisfiable.

Concurrency analysis. For concurrency analysis of a given program, **CM (%)** is the overall percentage of methods that could be parallelized with at least one other method (including itself) in class **C**. **CMP (%)** is the percentage of total number of method pairs that can be parallelised (including the method with itself) over all possible method pairs in a given program. For example, for class `ArrayCollection` in Listing 6 with 7 methods, the analysis shows that 6 out of 7 (85%) methods could potentially be executed with at least one other method in the class. For CMP(%), 13 pairs(46%) from a state space of a total of $\binom{7}{2}+7=28$ pairs of methods in this class can run in parallel.

For the monetcarlo program, the **CM** ratio is 76% and the **CMP** ratio is 50% for 17 classes. We observed that the exclusion of overloaded constructors does not affect the potential for concurrency as constructors cannot be parallelized with any other method during program execution. In summary, the permission-based concurrency analysis in terms of number of method pairs (CMP%) vary from 10 to 59% for the Java Grande Benchmark, and from 14 to 71% for Æminium and Plaid. Overall, the results show considerable potentials

Table 2: A brief statistics of benchmark programs, the analysis by Pulse, and annotation overhead. For each program **P** in the behchmark **B**, with the number of classes and the number of methods **M**, **SM**, **USM**, **CM(%)** and **CMP (%)** shows the number of methods Pulse identifies as satisfiable, unsatisfiable, percentage of potentially concurrent methods and percentage of concurrent method pair. **LOA**$_P$ and **Anns**$_P$ are the number of annotated lines in the Pulse translated version and the number of individual annotations respectively. **Anns**$_F$ is the number of permission annotations generated at field level. **QA** shows the quality analysis of inferred annotations for the Pulse translated version and **ES** (sec) shows the execution speed of `Sip4J` permission inference for a program, measured in seconds. ✠ It includes three inner classes. ▽ Overloaded methods including constructors. † case studies (java version) common in both benchmarks.

Statistics				Pulse analysis				Effectiveness analysis				
B	**P**	**C**	**M**	**SM**	**USM**	**CM (%)**	**CMP%**	**LOA**$_P$	**Anns**$_P$	**Anns**$_F$	**QA**	**ES** (sec)
Pulse	Pulse	40✠	434	416	18▽	371 (85%)	53%	513	1,471	5,352	24	342.40
Java Grande Benchmark	montecarlo	17	168	151	17	128 (76%)	50%	185	630	1,360	13	57.95
	search	8	33	33	0	18 (54%)	10%	42	118	455	0	1.71
	moldyn	7	25	25	0	17 (68%)	14%	33	103	478	0	1.47
	euler	7	34	34	0	25 (73%)	28%	42	123	742	4	17.31
	crypt	5	24	24	0	17 (70%)	54%	30	84	180	1	1.32
	lufact	5	25	25	0	18 (72%)	59%	31	127	246	0	2.40
	series	5	20	20	0	13 (65%)	40%	27	46	96	0	0.68
	sor	5	18	18	0	10 (55%)	46%	24	65	110	0	0.42
	sparsematmult	4	17	17	0	10 (58%)	53%	22	75	186	0	0.34
Æminium	blacksholes	6	34	32	2	22 (64%)	61%	41	157	194	2	7.85
	health	6	18	18	0	12 (66%)	18%	25	65	350	1	1.14
	gaknapsack	6	21	21	0	15 (71%)	40%	28	86	320	0	1.71
	quicksort	3	9	9	0	5 (55%)	37%	13	33	16	0	0.39
Plaid& Æminium †	fft	4	11	11	0	3 (27%)	14%	16	44	44	0	0.48
	shellsort	2	6	6	0	5 (83%)	67%	11	32	44	0	1.17
	webserver	3	12	12	0	9 (75%)	71%	16	25	16	0	1.02
	fibonacci	1	4	4	0	3 (75%)	60%	6	15	8	0	0.06
	integral	1	5	5	0	4 (80%)	67%	7	17	18	1	0.23
Plaid	deltablue	12	72	72	0	51 (70%)	59%	85	229	562	2	18.74
Example	ArrayCollection	2	9	9	0	7 (77%)	45%	12	36	22	0	0.19
Total		**149**	**999**	**962**	**37**	**763 (76%)**	-	**1,094**	**3,598**	**10,791**	**48**	**451.21**

in our inferred specifications to enable concurrency in sequential programs, which is one of the motivations of our work.

6.2. Effectiveness analysis of `Sip4J`

In this section, we demonstrate the effectiveness of our permission inference technique, in terms of the amount of annotations generated and the execution speed of the inference process.

6.2.1. Annotation overhead

We measure the annotation overhead as a way to quantify manual annotation effort by measuring the (1) the amount of annotations generated by `Sip4J` at field level (including parameters), (2) the number of annotated lines generated by `Sip4J` for Pulse translated version and (3) the number of individual annotations for the Pulse translated version using a single typestate. To compute annotation overhead, we use the number of methods **M**, as the basis for evaluation as we infer method-level permissions in a Java program.

LOA$_P$ captures the number of lines of annotations in the Pulse translated version of a program with **M** methods. **LOA**$_P$ counts one line (permission contract) for each method with a '`requires`' and an '`ensures`' clause, one line for each class to define typestate information at the class level, and one line to import the package that supports Plural annotations in a Java program. Therefore $\mathbf{P\_AL} = \mathbf{N} + \mathbf{C} + 1$.

Anns$_P$ calculates the number of Pulse annotations for **M** methods in the Pulse version of the program. The number of annotations in this case depends on the number of class fields and parameters accessed in a method. The metrics is calculated as follows.

For each *non-constructor method*, **Anns**$_P$ counts two (as part of '`requires`' and '`ensures`' clause) to define a typestate '`alive`' at the method level.

For each *method that accesses some class fields*, **Anns**$_P$ adds two (pre- and post-permission) for the current object `this`.

For each *method that also has parameters*, **Anns**$_P$ adds two (pre- and post-permission) for each parameter. Finally, **Anns**$_P$ 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 and one typestate annotation as constructor methods can have only '`ensures`' clause as part of the permission contract in Pulse as Pulse does not support overloaded constructors with parameters.

Let M be the set of methods, $\mathbf{M}_C$ be the number of constructors, $\mathbf{M}_{NC}^{F}$ be the number of non-constructor methods that access some class fields, and $\mathbf{M}_{NC}^{NF}$ be the number of non-constructor methods that do not access any class field (i.e., $\mathbf{M} = \mathbf{M}_C + \mathbf{M}_{NC}^{F} + \mathbf{MC}_{NC}^{NF}$). **Anns**$_P$ is defined as:

$$\mathbf{Anns}_P = (1+1)*\mathbf{M}_C + (2+2)*\mathbf{M}_{NC}^{F} + 2*\mathbf{M}_{NC}^{NF} + \sum_{m \in M}(2*P(m)) + \mathbf{C} \qquad (2)$$

where $P(m)$ denotes the number of parameters in m.

Anns$_F$**:** It calculates the number of permission-based annotations (pre- and post-permissions) generated at the class field level. Therefore, the number of permission-based annotations generated for all **M** methods is $\mathbf{Anns}_F = \sum_{m \in M}(2*RV(m))$, where $RV(m)$ is the number of referenced variables (class fields including parameters) accessed in method m. For example, for

the Pulse translated version of `ArrayCollection` program (Listings 6 & 9), `Sip4J` generated 12 annotated lines for 9 Methods in 2 classes with a total of 36 annotations at object level and 22 annotations (Listings 5 & 8) when permissions were generated at fields level.

The analysis shows the effectiveness of our inference technique. Otherwise, this amount of annotations would have to be manually created, which represents a significant overhead.

6.2.2. Quality Analysis of Inferred Specifications

To assess the quality of the inferred annotations, we annotated all programs in our dataset by manually generating permission annotations for the Pulse translated version of the same program. **QA** in Table 2 is the number of safe approximations made for the inferred annotations in each program compared to its manual counterpart. A non-zero number indicates the situation where `Sip4J` does not produce the optimal solution and the inferred annotations deviate from their manual counterparts. This will be discussed in Section 7). For example, in the case of Pulse, `Sip4J` generated 24 safe approximations for the inferred annotations. This occurs either when a referenced variable `x` is in a complex infix expression with nested prefix or postfix expressions or `x` is mapped with more than one parameter in a method call such as `foo(x, x)`. For the latter, we generate a single, safe permission for variable `x` to show the method's combined effect on it.

We observed that generating restrictive and 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.

6.2.3. Execution speed of permission inference

We compute efficiency of the permission inference algorithm by measuring the average execution time **ES** in seconds, averaged over 10 independent runs on Java Virtual Machine. The result shows that in total, `Sip4J` took 451 seconds (9 minutes) to parse and generate 1,094 permission contracts for 999 methods in 149 classes.

The result also shows that it takes 58 seconds (1 minute) to infer permission-based specifications for montecarlo in the Java Grande Benchmark with 168 methods, and a fraction of a second to a maximum of 19 seconds for programs in the Æminium and Plaid benchmarks. For Pulse, the biggest case study we evaluated with 10K plus lines of code, it took 342.40 seconds (6 minutes) to identify dependencies and generate 513 annotated lines with 5,352 annotations for 434 methods. If created manually, the above annotations may take multiples of months as in the case of Pulse [21][11]. It shows that the complexity and the design of the input program does have an impact on the annotation generation time. However, we expect that the execution time of `Sip4J` can be a multiple of minutes (or hours in the worse-case scenario for really large applications), but not in months, thus showing the effectiveness of our technique in generating permission-based annotations.

[11] In Pulse, authors reported nine months were spend on a Java application with 55 classes and 376 methods, for identifying and manually annotating it with 546 permission contracts for verification purpose.

6.3. Effectiveness of the inferred specifications

We analyse the effectiveness of the inferred permissions on five benchmark programs[12] by measuring their runtime in milliseconds (ms), averaged over 15 independent runs on Plaid. Plaid [7] is a new permission-based programming paradigm that verifies and parallelises execution of typestate-based Plaid program based on access permissions. Plaid borrows its grammar and lexical structure from Java Specification Language (JSL) but shows significant differences from Java to incorporate permission-based specifications as part of the language. However, it provides interoperability with Java programs. We can call Java methods on Plaid infrastructure by developing wrapper methods in Plaid. Every type in Plaid is explicitly represented as a tuple having a type structure and associated permission to show aliasing and mutability of the corresponding object type. Plaid supports three types of access permissions: unique and immutable for individual objects, and share permissions for shared variables.

To analyse the effectiveness of the inferred permissions, we developed wrapper methods in Plaid and called the Java methods by *lifting* the Plaid program with the permissions inferred by `Sip4J` for the Java version. We compared the execution speed of the input Java programs with their Plaid counterparts, both with and without annotations. We replaced pure permission with immutable and full with unique, as a conservative and safe approximation. The dataset for performance evaluation consists of five commonly-used recursive benchmark programs: fft, shellsort, integral, fibonacci, and webserver, from Æminium and Plaid. The average execution time of different versions of the same program along with their input configurations is given in Table 3. The u-JPlaid version was developed to demonstrate the difference in performance between two execution frameworks, i.e., native Java and Plaid, for the same Java program without annotations. The following observations can be made:

Table 3: Configurations and average execution times in ms of the four different versions of the benchmark program. ⋆ represents .java version of the original Java source program, compiled and run in Java Virtual Machine. § represents .plaid version of the source program developed using plaid wrapper methods around u-Java, compiled and run on Plaid. ⋈ represents the annotated .plaid version of the Java program using inferred permissions by `Sip4J`, compiled and run on Plaid. ◁ represents the annotated .plaid version of the same program present in the Plaid benchmark, compiled and run on Plaid.

Program	**Input Size**	**Type**	**u-Java⋆**	**u-JPlaid§**	**a-JPlaid⋈**	**a-Plaid∗**
fft	$n = 2^{15}$	Recursive	592.7	596.7	614.8	118,665.0
shellsort	$n = 10^5$	Recursive	1,692.3	1,811.1	1,754.7	106,617.7
integral	epsilon = 10^{-6}, x1 = 0.0, x2 = 1.0	Recursive	74.2	94.2	95.8	201,457.4
fibonacci	n = 25	Recursive	1.4	48.3	56.2	13,253.2
webserver	-	Iterative	16	45.5	45.7	53.6

- Permission inferred by `Sip4J` (column **a-JPlaid**) yields very similar runtime performance to that produced by the unannotated version of the same program (column **u-JPlaid**) on the Plaid infrastructure for all programs. In three of the five programs both

[12] when .plaid version of the source program was available in the Plaid benchmark. We are not able to evaluate performance of the rest of the benchmark programs due to unavailability of the .plaid version that otherwise need to be generated automatically or manually and that is not the immediate objective of this work.

versions show similar performance by running the same program in the Java Virtual Machine (column **u-Java**). For the last two programs the original Java version is faster and it takes at most 56 ms to complete its execution.

- Manually annotated Plaid version (column **a-Plaid**) of the same Java program takes significantly longer to execute than all others. The difference in performance can be attributed to Plaid's complex type system and runtime to support access permissions as first class language constructs [8], which actually increases the execution time, even for unannotated sequential Java versions (**u-JPlaid** and **a-JPlaid**). However, it does not invalidate the effectiveness of our technique, as any improvement gained on the Plaid infrastructure can be automatically exploited by our framework.

The results show that our framework can generate permission annotations that yield an execution performance competitive to their unannotated and manually annotated counterparts on the Plaid infrastructure, hence demonstrating its effectiveness.

7. Limitations

We have identified some limitations of the static analysis performed by `Sip4J`. Our technique generates safe permissions i.e., unique or full on the reference objects in the case of class library (APIs) method calls, due to the unavailability of library method definitions in the source code. The technique generates restrictive permissions (`share` in `Context-RW` and `full` in `Context-R`) for all reference variables accessed in a complex expression having a mix of prefix and postfix expressions. This is because, at the moment we do not parse individual expressions with or without pre- and postfix operator in a complex expression. We believe this is an engineering problem and generating restrictive permissions will not affect the integrity of the program during verification as permission-based specifications impose their own ordering constraints to avoid concurrent access by multiple references. Our technique is based on static analysis of source code and therefore generates safe permissions for the reference variables accessed inside dynamic expressions such as switch `cases`, nested if-else and others. At the moment, we do not handle dynamic method calls. However, as the proposed analysis is flow sensitive, in an assignment statement, we can find the object type a reference (left hand side) refers to by tracking the type of expression on the right hand side. This information can eventually be used to decide the actual overridden method to be parsed without executing the program. Our analysis also does not cover some constructs of in the Java Specification Language such as generics and lambda expressions.

8. Related Work

For the sake of brevity, we will discuss here the most relevant related work.

Plural is an extension of Java, implemented as an Eclipse plug-in [2–5]. It verifies correctness of sequential and concurrent object protocols such as Java APIs, based on access permissions. It performs intra-procedural static analysis of the annotated source code to identify and track permissions for every program variable (method parameters, the method receiver object and local variables) at every program point and issues warning for protocol violations in the program. It also infers permissions flow through the system to avoid the permission tracking overhead associated with splitting and joining the fractional permission

during program verification [6]. It helps programmers to statically follow API protocols without actually executing the program. However, permission-based typestate specifications are explicitly added as a part of method contract.

JavaSyp (Symbolic Permissions for efficient static program verification) [25] is a permission-based automated program verification tool that verifies the functional correctness of realist sequential programs such as Java array list with its iterator. The underlying approach combines symbolic permissions (unique and immutable) with JML contracts to enforce and control aliasing in the program. It performs static analysis of the permission-annotated source program to generate verification conditions (VCs). The program verification is performed based on inferred conditions. In this approach, permission tracking is straightforward as tracking symbolic values is much easier than fractions of values.

Pulse [19] is an automatic formal verification tool that soundly verifies the Plural annotated Java program but to exploit the functionality of Pulse, programmers manually add Plural specifications at method level.

VeriFast [18] is a sound, modular and a prototype automatic program verification tool for single and multi-threaded C and Java programs. VeriFast supports fractional permissions in the range (0,1]) to specify access rights for the heap locations in a program. To enable verification, programmer defines lemma functions to interactively specify permission-based contracts written in Separation Logic. The logic-based specifications are then tracked in the system to verify that pre-conditions imply post-conditions and proper termination of the function.

Viper [15] is a permission-based verification infrastructure for sequential, object-based intermediate language. It verifies heap structures in the program based on permissions contracts and the loop invariants. In Viper, programmer specifies permission contracts as accessibility predicate as `acc(e.f)`. A method can access a particular heap location `e.f` if appropriate permission are held by that location. Like Pulse, program verification is based on method specifications rather than its implementation.

Leino et al. [9] presented Chalice, a verification methodology to reason about shared state interference in multi-threaded programs. The proposed approach uses concept of Boyland's fractional permissions. It defines permission percentages as 'Full', 'Some' and 'No', in the range 0-100, to specify permission contracts for concurrent reading, (un)sharing of objects among multiple threads for each heap location at method level. The annotation acc(o.f) represents 'Full' permission (100%) that shows that a thread has exclusive access to the field (f). A non-zero or 'Some' permission shows a read-only access to a particular location (o.f) that is represented by rd(o.f). The permission contracts are transferred from a monitor to a thread that acquires this monitor. The underlying approach ensures that the sum of permissions from all threads remain less than or equal to 100%. The proposed methodology is realized using a concurrent program verifier called Chalice [10]. Chalice does not support automatic inference of permission-based specifications but it uses autoMagic, a command-line option, to infer read and write accesses as 'access' and 'pure' assertions for non-heap locations. In Chalice, programmer manually annotates program with permission-based predicates and class invariants to verify its correctness.

Ferrara et al. [11] proposed a small object-based language to develop and verify correctness of concurrent programs. It uses access permissions to ensure thread non-interference for the shared heap locations in a program. The proposed approach performs static analysis of the source code based on abstract interpretations [31]. It computes symbolic values for each heap location that are then used to generate permission-based concurrent specifications in the

form of fractional permission and counting permission but to infer access notations in this way, programmer manually annotate program with pre and post conditions at method level. In an extended work, Dohrauet et al. in [14] proposed a static analysis to infer permission-based contracts for array manipulating concurrent programs. The idea is to associate separate permission with each array element based on Separation Logic [32]. For this purpose, the technique employs fractional permission to ensure the exclusive (full) write access on a particular location while allowing the concurrent (read) access using any positive fraction of permission.

It is generally acknowledged that specifications based on fractions (concrete) values are tedious to write and harder to track and adapt for programmers [12].

Heule et al. [12] proposed a technique that automatically converts fractional permissions for shared-memory concurrent programs into abstract read and write permissions. The objective was to avoid complex reasoning overhead to specify concrete (fractional) values for concurrent constructs in a program. This permission system generates two kinds of access notations i.e., *full* and *read*, where *full* represents exclusive access rights and *read* represents the read-only access to a shared variable. Like other approached, the proposed approach takes an annotate program with accessibility predicates (access & read) as a part of method specifications to infer permission.

Having a language runtime support for permissions in Plaid [7], Stork et al. proposed a new programming paradigm Æminium [8][13], to parallelise execution of sequential programs based on access permissions. The system supports three kinds of access permissions: unique, immutable, and share. Æminium is able to achieve significant performance improvements, but at the cost of increasing programmers' burden to explicitly specify statefull effects in the form of permission-based dependency information at method level.

Unlike previous work, Sadiq et al [33] proposed a preliminary idea to automatically infer symbolic permissions from the source code of Java program, the most commonly used programming language. In our proposed work, we extend on this idea and present a comprehensive framework (permission inference technique and tool) to automatically reveal permission-based implicit dependencies from sequential Java programs at method level, without using any intermediate representations and method-level annotations. The inferred permissions such as unique, full etc., can help programmers to work at a higher level of abstractions and to avoid the significant annotation and reasoning overhead associated with concrete values. We feel that relieving programmers from the complex specification (annotation) overhead, the aim of our research, is applicable to both explicit and implicit concurrency approaches and necessary to increase the wider adoption of these approaches.

9. Conclusion

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. However, such constructs give rise to significant code complexity and errors. Therefore, permission-based contracts have been used to formally verify the correctness of software applications. Moreover, access permission-based specifications have

[13] `https://github.com/AEminium/AeminiumBenchmarks/tree/master/src/aeminium/runtime/benchmarks/`

been proposed as an alternative approach to achieve implicit concurrency without explicitly introducing concurrency constructs. However, these specifications need to be manually added in the source program imposing significant additional work for programmers which may result in more errors.

It has been observed and acknowledged in literature that manually annotating a small program with permission specifications can be quite easy for a human. However, as the size and complexity of program grows, it becomes increasingly challenging to track data flow and alias flow information manually across the program, which creates the need for an automatic tool to identify such subtle dependencies correctly and in an efficient manner. In this paper, we propose, `Sip4J`, a framework of automatically inferring permission-based implicit dependencies in the form of symbolic permissions from sequential Java programs by performing static analysis of the source code. Moreover, we integrated Pulse, a permission-based verification tool as a part of our proposed framework to perform empirical evaluation of the inferred specifications and to reason about the concurrent behavior of the sequential programs.

Experiments were performed on 20 programs with 999 methods from three widely-used benchmark datasets, Java Grande Benchmark, Æminium, Plaid and Pulse itself to evaluate the correctness and effectiveness of the inferred annotations, as well as the efficiency of our inference algorithm. The concurrency analysis of the inferred specifications using Pulse confirms that on average, there are potentials to parallelise the execution of 763 (76%) methods in the benchmark programs. Our algorithm can also infer permission-based annotations efficiently, averaging 0.45 seconds for annotating a method.

The experiments demonstrate the feasibility and benefits of our framework to the related permission-based verification and parallelization approaches in the literature to achieve the intended benefits without additional work imposed on programmers. As access permissions impose their own ordering constraints, this information along with the control flow information of a program can be used to automatically parallelise the execution of sequential Java program without manual annotation effort and without the need to learn any new permission-based specification/programming languages. The inferred specifications can also be used to institute concurrency control in multi-threaded programs.

We have envisaged a number of future directions for our framework. We plan to (a) expand static analysis to incorporate more Java language constructs such as polymorphism, generics, lambda expressions and others; (b) infer permissions at a more granular level, such as individual permissions for members of collections or arrays; (c) develop an online system to encourage wider adoption of the proposed technique; (d) automatically infer the permission-based concurrent specifications such as monitor invariants for concurrent programs; and (f) extend Pulse analysis to overcome its existing limitations and provide a comprehensive support for the Java Specification Language.

References

[1] J. Boyland, Checking interference with fractional permissions, in: SAS 2003, USA, 2003, pp. 55–72.

[2] N. E. Beckman, K. Bierhoff, J. Aldrich, Verifying correct usage of atomic blocks and typestate, in: Proceedings of the 23rd ACM SIGPLAN Conference, OOPSLA '08, 2008, pp. 227–244.

[3] K. Bierhoff, J. Aldrich, Modular typestate checking of aliased objects, in: Proceedings of the 22Nd Annual ACM SIGPLAN Conference, OOPSLA '07, 2007, pp. 301–320.

[4] K. Bierhoff, J. Aldrich, Plural: Checking protocol compliance under aliasing, in: ICSE Companion '08, 2008, pp. 971–972.

[5] N. 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, ACM, 2009, pp. 737–738.

[6] K. Bierhoff, N. E. Beckman, J. Aldrich, Polymorphic fractional permission inference (2009).

[7] J. Aldrich, R. Bocchino, R. Garcia, M. Hahnenberg, M. Mohr, K. Naden, D. Saini, S. Stork, J. Sunshine, É. Tanter, et al., Plaid: a permission-based programming language, in: OOPSLA'11, 2011, pp. 183–184.

[8] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, J. Aldrich, Aeminium: A permission-based concurrent-by-default programming language approach, TOPLAS, 36 (1) (2014) 1–42.

[9] K. R. M. Leino, P. Müller, A basis for verifying multi-threaded programs, in: ESOP, Springer, 2009, pp. 378–393.

[10] K. R. M. Leino, P. Müller, J. Smans, Verification of concurrent programs with chalice, in: FOSAD 2007/2008/2009 Tutorial Lectures, Springer Berlin Heidelberg, 2009, pp. 195–222.

[11] P. Ferrara, P. Müller, Automatic inference of access permissions, in: VMCAI'12, USA, 2012, pp. 202–218.

[12] S. Heule, K. R. M. Leino, P. Müller, A. J. Summers, Abstract read permissions: Fractional permissions without the fractions, in: VMCAI'13, 2013, pp. 315–334.

[13] J. T. Boyland, P. Müller, M. Schwerhoff, A. J. Summers, Constraint Semantics for Abstract Read Permissions, in: FTfJP'14, ACM, 2014, pp. 1–6.

[14] J. Dohrau, A. J. Summers, C. Urban, S. Münger, P. Müller, Permission Inference for Array Programs. URL `https://arxiv.org/abs/1804.04091`

[15] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, A. J. Summers, Viper, Tech. rep. (2014). URL `https://www.research-collection.ethz.ch/handle/20.500.11850/85086`

[16] P. Müller, M. Schwerhoff, A. J. Summers, Viper: A verification infrastructure for permission-based reasoning, in: Dependable Software Systems Engineering, 2017.

[17] J. Wickerson, M. Dodds, M. Parkinson, Explicit stabilisation for modular rely-guarantee reasoning, in: Programming Languages and Systems, Springer Berlin Heidelberg, 2010, pp. 610–629.

[18] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: A powerful, sound, predictable, fast verifier for C and \java, in: NASA Formal Methods Symposium, 2011, pp. 41–55.

[19] R. I. Siminiceanu, I. Ahmed, N. Cataño, Automated verification of specifications with typestates and access permissions, ECEASST (2012) 1–15.

[20] J. Wickerson, Concurrent verification for sequential programs (2013).

[21] N. Cataño, I. Ahmed, R. I. Siminiceanu, J. Aldrich, A case study on the lightweight verification of a multi-threaded task server, Sci. Comput. Program. 80 (2014) 169–187.

[22] B. Jacobs, D. Bosnacki, R. Kuiper, Modular termination verification of single-threaded and multithreaded programs, ACM TOPLAS 40 (3).

[23] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1) (1987) 1 – 101.

[24] B. Meyer, Applying "design by contract", Computer 25 (10) (1992) 40–51.

[25] K. Bierhoff, Automated program verification made symplar: symbolic permissions for lightweight automated reasoning, in: SIGPLAN, Onward! 2011.

[26] K. Bierhoff, N. E. Beckman, J. Aldrich, Practical API protocol checking with access permissions, in: ECOOP‘09, Genoa, 2009, pp. 195–219.

[27] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[28] R. E. Strom, S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE TSE (1) (1986) 157–171.

[29] R. S. P. Roux, Model checking with edge-valued decision diagrams, NFM, NASA/CP-2010-216215 (April, 2010) 222–226.

[30] L. A. Smith, J. M. Bull, J. Obdrizalek, A parallel java grande benchmark suite, in: ACM/IEEE, SC ’01, 2001, pp. 6–6.

[31] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: POPL’77, ACM, 1977, pp. 238–252.

[32] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, IEEE, 2002, pp. 55–74.

[33] A. Sadiq, Y. Li, S. Ling, I. Ahmed, Extracting permission-based specifications from a sequential java program, in: ICECCS‘16, 2016, pp. 215–218.

Appendix A. Client Classes for the ArrayCollection Example

This section shows the client classes for both annotated and unannotated versions of the input Java program with `ArrayCollection` class given in Listing 4. Listing 7 shows the `ArrayClient` class for the unannotated version of `ArrayCollection` class. The `ArrayClient` class has `main()` method that acts as a client method for all the methods of `ArrayCollection` class.

Listing 7: The client program for the un-annotated ArrayCollection class given in Listing 4.

```
class ArrayClient{
 public static Integer[] e1 = new Integer[10];
 public static void main(String[] args) {
    ArrayCollection box = new ArrayCollection(10);
    box.initColl();
    box.displayColl();
    box.initE(e1);
    box.copyColl();
    box.displayE(e1);
    box.tidyupColl();}
}
```

Listing 8 shows an annotated version of the `ArrayClient` class with permission-based specifications generated at class field level by `Sip4J`.

Listing 8: The permission annotated version of ArrayClient class given in Listing 7.

```
package outputs;
class ArrayClient{
 //none(e1)—∘ unique(e1)
 ArrayClient(){ }
 //none(coll) * none(e1)—∘ unique(coll) * unique(e1)
 public static void main(String[] args){ }
 }
```

Listing 9 shows a Plural annotated version of `ArrayClient` class generated by `Sip4J` with access permission defined on the receiver object to perform verification by Pulse.

Listing 9: The Pulse annotated version of ArrayClient class given in Listing 8.

```
@States({@State(name = "alive")})
class ArrayClient {
@Perm(ensures="unique(this) in alive")
ArrayClient() {}
@Perm(requires="none(this) in alive",ensures="unique(this) in alive")
public static void main(String[] args) {
}
}ENDOFCLASS
```

Appendix B. `Sip4J` rules

This section provides mathematical rules to construct a graph model from the extracted information during parsing. The rules are organised into subsections by their usage in the `Sip4J` framework such as a) *Context* rules b) *Statement rules* and c) the *permission inference* rules. Appendix B.1 shows the context rules that models the read, write behavior of other methods on the reference variables (`<grv>`, `<lrv>`). Appendix B.2 shows the graph construction rules for expressions capturing global references (`<grv>`). The graph construction rules for the expressions manipulating local references (`<lrv>`) are shown in Appendix B.3. The graph construction rules for method calls expressions are given in Appendix B.4.

Appendix B.1. ***Context rules***

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

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

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

$$\frac{\texttt{<lrv>}}{(\exists \texttt{aliasEdge(<lrv>,<grv>)} \Longrightarrow \texttt{apply(Context-R,<grv>)})}\ \textbf{(Context-R, <lrv>)}$$

$$\frac{\texttt{<lrv>}}{(\exists \texttt{aliasEdge(<lrv>,<grv>)} \Longrightarrow \texttt{apply(Context-RW,<grv>)})}\ \textbf{(Context-RW, <lrv>)}$$

$$\frac{\texttt{<lrv>}}{(\exists \texttt{aliasEdge(<lrv>,<grv>)} \Longrightarrow \texttt{apply(Context-N,<grv>)})}\ \textbf{(Context-N, <lrv>)}$$

Appendix B.2. ***Global references rules***

$$\frac{\texttt{<Type> <grv>}}{\texttt{<do-nothing>}}\ (\textbf{GR-Decl,}\ \texttt{<grv>})$$

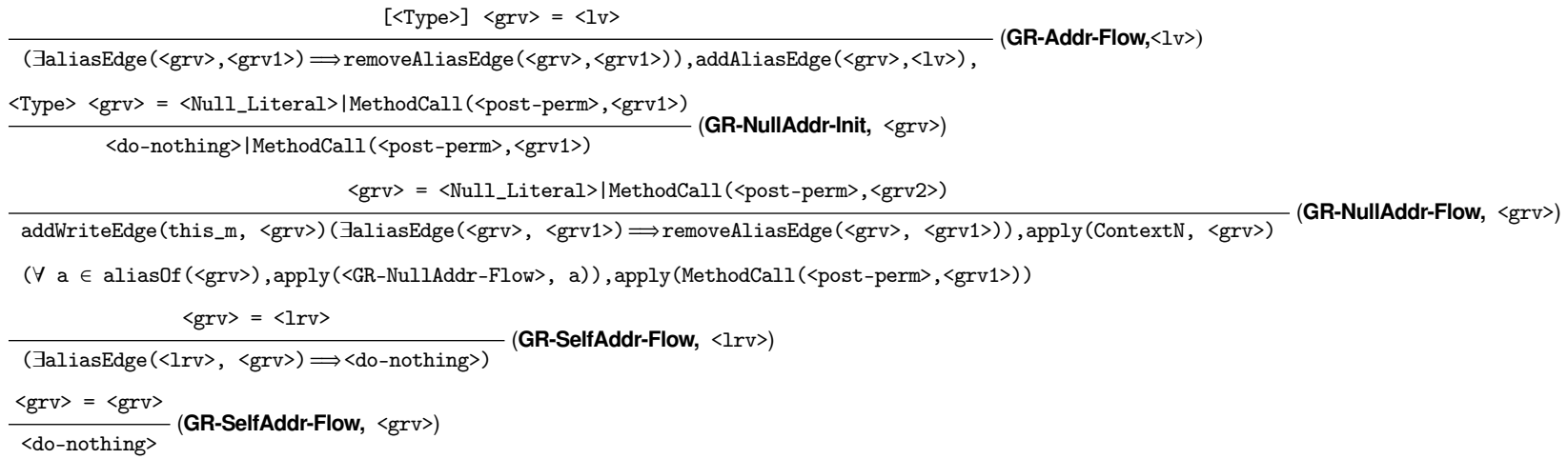
$$\frac{\texttt{<grv>}}{\texttt{addReadEdge(this\_m,<grv>)}}\ (\textbf{GR-Read-Only,}\texttt{<grv>})$$

$$\frac{\texttt{[Type] <grv> = <LITERAL>|<grv1> |MethodCall(<post-perm>,<grv1>)}}{\texttt{addWriteEdge(this\_m, <grv>)}(\forall a \in \texttt{aliasOf(<grv>) addWriteEdge(this\_m, a)),(apply(GR-Read-Only,<grv1>)}\ \texttt{|apply(MethodCall(<post-perm>,<grv1>))}}\ (\textbf{GR-Val-Flow,}\texttt{<grv>})$$

$$\frac{\texttt{[<Type>] <grv> = new <Type>(<grv2>)|<Literal>||MethodCall(<post-perm>,<grv2>)}}{\texttt{addWriteEdge(this\_m,<grv>),apply(Context-N,<grv>),(apply(GR-Read-Only,<grv2>)|apply(MethodCall(<post-perm>,<grv1>))}}\ (\textbf{GR-New-Obj,}\ \texttt{<grv>})$$

$$\frac{\texttt{<grv> = <grv1>}}{(\exists \texttt{aliasEdge(<grv>,<grv2>)} \Longrightarrow \texttt{removeAliasEdge(<grv>,<grv2>)),addAliasEdge(<grv>,<grv1>),apply(GR-Read-Only,<grv1>)}}\ (\textbf{GR-Add-Flow,}\ \texttt{<grv>})$$

$$\frac{\texttt{[<Type>] <grv> = <lrv>}}{(\exists \texttt{aliasEdge(<lrv>,<grv1>)} \Longrightarrow \texttt{addAliasEdge(<grv>,<lrv>))}\ (\exists \texttt{aliasEdge(<grv>,<grv2>)} \Longrightarrow \texttt{removeAliasEdge(<grv>,<grv2>)),apply(GR-Read-Only, <grv1>)}}\ (\textbf{GR-Addr-Flow,}\ \texttt{<lrv>})$$


[<Type>] <grv> = <lv>
(∃aliasEdge(<grv>,<grv1>) ⟹ removeAliasEdge(<grv>,<grv1>)),addAliasEdge(<grv>,<lv>),
(GR-Addr-Flow,<lv>)
<Type> <grv> = <Null_Literal>|MethodCall(<post-perm>,<grv1>)
<do-nothing>|MethodCall(<post-perm>,<grv1>)
(GR-NullAddr-Init, <grv>)
<grv> = <Null_Literal>|MethodCall(<post-perm>,<grv2>)
addWriteEdge(this_m, <grv>)(∃aliasEdge(<grv>, <grv1>) ⟹ removeAliasEdge(<grv>, <grv1>)),apply(ContextN, <grv>)
(∀ a ∈ aliasOf(<grv>),apply(<GR-NullAddr-Flow>, a)),apply(MethodCall(<post-perm>,<grv1>))
(GR-NullAddr-Flow, <grv>)
<grv> = <lrv>
(∃aliasEdge(<lrv>, <grv>) ⟹ <do-nothing>)
(GR-SelfAddr-Flow, <lrv>)
<grv> = <grv>
<do-nothing>
(GR-SelfAddr-Flow, <grv>)

Appendix B.3. ***Local reference rules***

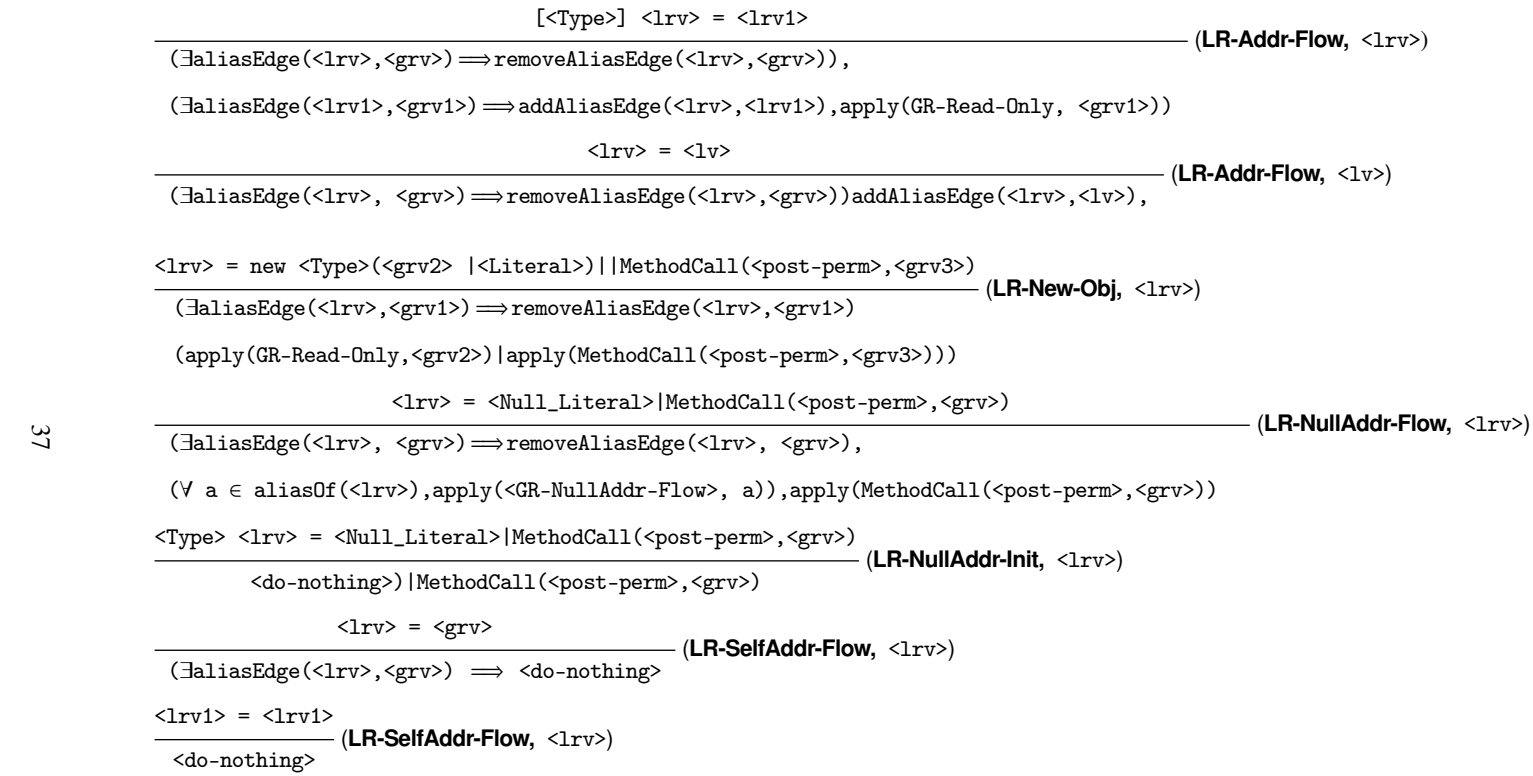
$$\frac{\texttt{<Type> <lrv>}}{\texttt{<do-nothing>)}}\ (\textbf{LR-Decl,}\ \texttt{<lrv>})$$

$$\frac{\texttt{<lrv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \Longrightarrow \texttt{(apply(GR-Read-Only(<grv>))}}\ (\textbf{LR-Read-Only,}\ \texttt{<lrv>})$$

$$\frac{\texttt{<lrv> = <Literal>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv>)} \Longrightarrow \texttt{apply(GR-Val-Flow, <grv>))}}\ (\textbf{LR-Val-Flow,}\ \texttt{<lrv>})$$

$$\frac{\texttt{[<Type>] <lrv> = <grv>}}{\texttt{(}\exists\texttt{aliasEdge(<lrv>, <grv1>),removeAliasEdge(<lrv>,<grv1>),addAliasEdge(<lrv>,<grv>))}}\ (\textbf{LR-Addr-Flow,}\ \texttt{<grv>})$$

`apply(GR-Read-Only, <grv>)`


[<Type>] <lrv> = <lrv1>
(∃aliasEdge(<lrv>,<grv>) ⟹ removeAliasEdge(<lrv>,<grv>)),
(∃aliasEdge(<lrv1>,<grv1>) ⟹ addAliasEdge(<lrv>,<lrv1>),apply(GR-Read-Only, <grv1>))
(LR-Addr-Flow, <lrv>)
<lrv> = <lv>
(∃aliasEdge(<lrv>, <grv>) ⟹ removeAliasEdge(<lrv>,<grv>))addAliasEdge(<lrv>,<lv>),
(LR-Addr-Flow, <lv>)
<lrv> = new <Type>(<grv2> |<Literal>)||MethodCall(<post-perm>,<grv3>)
(∃aliasEdge(<lrv>,<grv1>) ⟹ removeAliasEdge(<lrv>,<grv1>)
(apply(GR-Read-Only,<grv2>)|apply(MethodCall(<post-perm>,<grv3>)))
(LR-New-Obj, <lrv>)
<lrv> = <Null_Literal>|MethodCall(<post-perm>,<grv>)
(∃aliasEdge(<lrv>, <grv>) ⟹ removeAliasEdge(<lrv>, <grv>),
(∀ a ∈ aliasOf(<lrv>),apply(<GR-NullAddr-Flow>, a)),apply(MethodCall(<post-perm>,<grv>))
(LR-NullAddr-Flow, <lrv>)
<Type> <lrv> = <Null_Literal>|MethodCall(<post-perm>,<grv>)
<do-nothing>)|MethodCall(<post-perm>,<grv>)
(LR-NullAddr-Init, <lrv>)
<lrv> = <grv>
(∃aliasEdge(<lrv>,<grv>) ⟹ <do-nothing>
(LR-SelfAddr-Flow, <lrv>)
<lrv1> = <lrv1>
<do-nothing>
(LR-SelfAddr-Flow, <lrv>)

Appendix B.4. ***Method call rules***

$$\frac{\texttt{MethodCall(<Immutable>, <grv>)}}{\texttt{addReadEdge(<grv>,this\_m) apply(Context-R, <grv>)}}\ (\textbf{MethodCall(Immutable,}\ \texttt{<grv>})\textbf{)}$$

$$\frac{\texttt{MethodCall(<Pure>, <grv>)}}{\texttt{addReadEdge(this\_m,<grv>), apply(Context-RW, <grv>)}}\ (\textbf{MethodCall(Pure,}\ \texttt{<grv>})\textbf{)}$$

$$\frac{\texttt{MethodCall(<Full>, <grv>)}}{\texttt{addWriteEdge(this\_m,<grv>), apply(Context-R, <grv>)}}\ (\textbf{MethodCall(Full,}\ \texttt{<grv>})\textbf{)}$$

$$\frac{\texttt{MethodCall(<Share>, <grv>)}}{\texttt{addWriteEdge(this\_m,<grv>) apply(Context-RW, <grv>)}}\ (\textbf{MethodCall(Share,}\ \texttt{<grv>})\textbf{)}$$

$$\frac{\texttt{MethodCall(<Unique>, <grv>)}}{\texttt{addWriteEdge(this\_m,<grv>), apply(Context-N, <grv>)}}\ (\textbf{MethodCall(Unique,}\ \texttt{<grv>})\textbf{)}$$

$$\frac{\texttt{MethodCall(<None>, <grv>)}}{\texttt{<do-nothing>}}\ (\textbf{MethodCall(None,}\ \texttt{<grv>})\textbf{)}$$

Appendix B.5. ***Sip4J permission inference rules***

This section shows the graph traversal rules to generate permissions on the referenced variable <grv>.

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

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

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

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

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

$$\frac{\neg\exists \texttt{writeEdge(this\_m,<grv>)} \wedge\exists \texttt{readEdge(<grv>,this\_m)} \wedge\neg\exists \texttt{writeEdge(context,<grv>)} \wedge\neg\exists \texttt{readEdge(<grv>,context)}}{\texttt{none(<grv>)}}\ (\textbf{none})$$