Typestates and Beyond: Verifying Rich Behavioral Properties Over Complex Programs

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Ashish Mishra

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

August 2018

Declaration of Originality

I, **Ashish Mishra**, with SR No. **04-04-00-17-12-11-1-08637** hereby declare that the material presented in the thesis titled

Typestates and Beyond
Verifying Rich Behavioral Properties Over Complex Programs

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2011-2018**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discusions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Professor Y N Srikant Advisor Signature

DEDICATED TO

"the spirit of survival"

(most wonderful gift of evolution to humankind)

Acknowledgements

I am indebted to several individuals and groups who guided me, supported me, challenged me and contributed towards the completion of this thesis. First and foremost, I would like to thank my Ph.D. advisor Prof. Y. N. Srikant who showed immense confidence in me and provided me the opportunity to pursue this domain and perform this research. He was always available for uncountable technical as well as many non-technical discussions and was always patient with my unending series of questions.

I would like to thank Prof. Aditya Kanade who contributed to the first part of the thesis through numerous technical discussions and inputs. I owe a thanks to Prof. Deepak D'souza for his contributions and mentoring towards the later part of the thesis. He was always reachable and guided me to sail through, and comprehend related results and works in the domain of theory of computation.

I would like to thank the Department of CSA and IISc for providing an inspirational, productive environment. Many thanks to the CSA office staff for taking care of administrative matters quickly and efficiently. I am thankful to the MHRD, Government of India, for funding this research. I am thankful to my fellow students and members of the Compiler Lab who provided a friendly atmosphere and a place for numerous discussions. I would also like to thank all my friends at IISc, who were always there for me whenever I needed. Finally, a special thanks to my wonderful family for their immense understanding and constant support and encouragement, without them this work would not have been possible.

Abstract

Statically verifying behavioral properties of programs is an important research problem. An efficient solution to this problem will have visible effects over multiple domains, ranging from program development, program debugging, program correction and verification, etc. Type systems are the most prevalently used static, light-weight verification systems for verifying certain properties of programs. Unfortunately, simple types are inadequate at verifying many behavioral/dynamic properties of programs. Typestates can tame this inadequacy of simple types by associating each type in a programming language with a state information. However, there are two major challenges in statically analyzing and verifying typestate properties over programs.

The first challenge may be attributed to "increasing complexity of programs". The original work on typestates can only verify/analyze a typestate property over very *simple programs* which lacked dynamic memory allocation or aliasing. Subsequently, the following works on typestates extended and improvised the analysis over programs with aliasing and heaps. However, the state-of-the-art static typestate analysis works still cannot handle formidably rich programming features like asynchrony, library calls and callbacks, concurrency, etc. The second challenge may be attributed to "complexity of the property being verified". The original and the current notion of typestates can only verify a property definable through a finite-state abstraction. This makes the state-of-the-art typestate analysis and verification works inadequate to verify useful but richer non-regular program properties. For example, using classical typestates we can verify a property like, "pop be called on a stack only after a push operation", but we cannot verify a non-regular program property like, "number of push operations should be at least equal to the number of pop operations". Currently, these behavioral properties are mostly verified/enforced by programmers at runtime via explicit checks. Unfortunately, these runtime checks are costly, error-prone, and lay an extra burden on the programmer.

In this thesis we take small steps towards tackling both these challenges. Addressing complex program features, we present an asynchrony-aware static analysis, taking Android applications as our use case. Android applications have convoluted control flow, and complex features,

like asynchronous inter-component communications, library callbacks, Android enforced control flows (called as lifecycles), resource XMLs, which define and register event-handlers etc. Unfortunately, none of the available static analysis works for Android soundly captures all these features. We provide a formal semantics for Android asynchronous control flow, capturing these features. We use this semantics to introduce an intermediate program representation for Android applications, called the Android Inter-Component Control Flow Graph (AICCFG), and develop an asynchrony-aware interprocedural static analysis framework for Android applications. We use this framework to develop a static typestate analysis to capture Android *resource API* usage protocol violations. We present a set of benchmark applications for different resource types, and empirically compare our typestate analysis with the state-of-the-art synchronous static analyses for Android applications.

Addressing the challenges associated with increasing complexity of properties, we present an expressive notion of typestates called, Parameterized typestates (p-typestates). p-typestates, associate an extra Pressburger definable property along with states of regular typestates. This allows p-typestates to express many useful non-regular properties. We formally define this notion of p-typestates, and a p-typestate property automaton, to represent a p-typestate property. We present a *dependent type system* for these parameterized typestates and present a simple typestate-oriented language incorporating p-typestates. Further, typechecking such rich p-typestate properties require a programmer to provide invariants for loops and recursive structures. Unfortunately, providing these invariants is a non-trivial task even for expert programmers. To solve this problem, we present a simple and novel loop invariant calculation approach for Pressburger definable systems. We encode a number of real-world programs in our dependently-typed language and use p-typestates type system, and loop-invariant calculation to verify several rich properties which are not amenable to regular typestate analysis and verification.

Finally we discuss several possible extensions of our thesis along both these directions.

Publications based on this Thesis

- A. Mishra, A. Kanade and Y. N. Srikant, "Asynchrony-aware static analysis of Android applications," 2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), 2016.
- Ashish Mishra and Y. N. Srikant. "Analysis and verification of rich typestate properties for complex programs, ". ECOOP 2017 Doctoral Symposium, Barcelona, Spain, 2017.
- Ashish Mishra, Deepak D'Souza, Y. N. Srikant, "Presburger-Definable Parameterized Typestates, " - *Under-preparation.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growth of programming languages in any programming paradigm or field is guided by numerous factors and is mostly followed by a phase of development of theories, tools, techniques and algorithms for reasoning about the programs written in these language. Consider the case of *Object-oriented* languages. The first language to introduce the concept of *Objects* was Simula. The initial version of the language Simula-1 was introduced in 1966. The language was created to aid simulations, which typically modeled real world systems. Many of these systems contained hundreds and thousands of interacting parts. To model these systems and their interaction, Simula introduced the concept of Modules based not on procedures but on actual physical objects. Simula was followed by Smalltalk, which is considered the first true *Object-oriented* programming language. Smalltalk introduced many other new concepts like browsers, windows, menus and other GUI components which became major propelling factor for the object-oriented programming languages. Although Smalltalk gave Object-oriented development a certain amount of legitimacy in the marketplace, it took C++ to bring Object-oriented development what it really needed, widespread acceptance in the marketplace. The success of C++ gave way to the origins of Java in around 1992, a language to tap the growing market of consumer electronics, having Object-oriented principles at its core but only having those features of C++ which were "worthwhile". As these language and the paradigm matured and as their target fields expanded, they became bulkier and messier, with addition of newer language features, libraries, tools, frameworks, etc. Unfortunately, as a language and its ecosystem become more complex, it becomes harder to comprehend and debug the programs written in it and higher is the risk of developing erroneous programs with unwanted behaviors and functionalities. This may open up numerous vulnerabilities and attack surfaces which are potentially exploitable. Thus, parallel to their growth, there is also a growth of analysis tools, formal methods and verification techniques for these languages. The need to reason about

programs has been a guiding force for research in the field of programming languages, program analysis and program verification, program comprehension, etc. This has lead to great efforts towards developing various approaches for reasoning and verifying structural(static properties related with the structure) and behavioral(dynamic properties related with dynamic state) properties of programs.

Type systems offer one such approach which is used mainly for reasoning about structural properties of programs. Types classify data based on the valid operations over it. This defines the set of valid operations over data classes, checks if some operation or a sequence of operations is invalid and in the best case guarantees the absence of any unwanted/incorrect behavior.

Although types and type systems are good at modeling and verifying properties related to a fixed structure of data, they are not efficient mechanisms for describing and reasoning about dynamic aspects(behavioral properties) of programs. For instance, types can efficiently capture the operations defined over a datum, but cannot describe a stateful property describing a subset of these operations valid in a given state of the datum.

These dynamic aspects of programs and data are captured via a group of systems which fall under the purview or *Behavioral types* [61]. The most common example of systems under behavioral types include, *session types* and *typestates*. Typestates defines state-dependent availability or unavailability of operations, and is effective at modeling behavioral properties of programs, particularly, modeling and verifying protocols associated with data and programs. Although typestates are well understood, they have been applied to programs with limited features (simple programs with references) and regular properties (finite abstraction properties). This thesis makes theoretical and practical contributions towards extending typestates to richer properties and programs with complex features. This chapter, introduces the thesis and discusses the major contributions of the thesis.

1.1 Protocols and Other Typestate Properties

Protocols are ubiquitous. They act as an explicit or implicit contract between components of a software system or across systems. These contracts must be respected by all the software components that are party to the protocol. Two examples are a producer-consumer related protocol between two components acting as server and client respectively, and a communication protocol between the sender and the receiver. Often these protocols need not be a multi-party contract. They may be a set of rules associated with a data structure or an object, such as, "no use of a variable before its definition, or, "no reading from a closed File object". Violating these protocols or contracts could have effects ranging from being totally benign to semantically invalid programs, or in many cases opening up exploitable vulnerabilities in a program. For

example, "violating array access bounds" might lead to buffer-overflow vulnerabilities which might act as the root cause of numerous attacks.

Many of these properties and protocols cannot be adequately captured using normal *types* available in programming languages. This can be attributed to the inherent *non-transient* property of types, i.e., the type associated with a datum remains constant during its life. This makes these types inappropriate to capture transient/dynamic behavior of data. For instance, a simple *File* type can capture set of operations defined on a File like *read, write, open*, etc., but it cannot capture a dynamic property, which allows/disallows a certain subset of these operations based on the *state* of a File, or it cannot capture a property which is a function of the size of a File.

A *Typestate* [115] is programming concept useful for enforcing such protocols and properties over data and programs. They form a component of the general *Behavioral Types* [61] capturing the dynamic or behavioral properties of data as compared to regular *types* which capture the static structural properties of data. A classic example of typestates is a *FileManager* which allows a File object to have a set of operations defined over it, viz., *open, close, read and write.* A subset of these operations is valid on a File object based on its current state. Figure 1.1, shows a finite automaton representing the valid operations in open and closed states of a File object and pre- and postconditions for each method. Typestates are good at enforcing properties over states and state transformations for data or program. This makes them particularly useful to check or enforce interesting properties over imperative programs, which are basically transformers or functions over the state of data. These properties can be checked or enforced either statically or dynamically thereby making software reliable and robust by early elimination of many semantic errors in programs.

Many of the important rich safety properties are not definable using regular typestates. For instance, a property defined over a *Stack* as " the number of push operations over a stack is always greater than or equal to the number of pop operations over it". This property can be defined over a *Counter Automaton* but not using regular finite state machines. Fortunately, these properties can be verified using dependent types and can be statically typechecked by restricting the parameters of the dependent types to *Presburger arithmetic formulas.* Similarly, there may be a different logical family for other classes of properties which provides required expressiveness to type systems while still balancing on the decidability of typechecking and the amount of annotation required. A portion of this thesis discusses this limitation of regular typestates in detail and provides a generalized notion of typestates providing an adequate balance between expressiveness and decidability of typechecking.

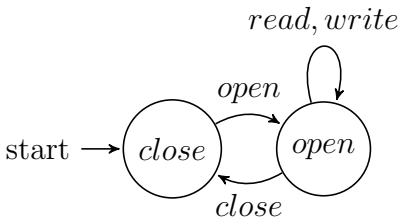


Figure 1.1: Typestate property for FileManager

1.2 Typestate Analysis and Verification

The problem of analyzing or enforcing a typestate property is termed as *typestate analysis*. The property could be enforced either statically or dynamically. There are various approaches in literature to verify and enforce typestate properties. This is achieved either using a static/dynamic program analysis, which analyzes the program to check for possible typestate violations or proves their absence or using a more language-based approach where typestates are integrated as a language feature. These languages are called *typestate-oriented* languages. In this thesis, we make theoretical and practical contributions to both these approaches.

In the absence of the above mentioned typestate analysis approaches, protocols and other typestate properties are generally specified as *class documents* or other documentation associated with code. These documentations contain an informal "finite state machine" defining the set of valid operations/methods in each state of the data/object. The programmer then needs to enforce this property at runtime. This ad-hoc approach of dynamically enforcing typestate properties is error prone and may lead to typestate violations. Debugging these programs is hard since the property being enforced is not a part of the program, thereby making it hard to comprehend the error. Further, the error generated is separated from the source of error, both in terms of space and time.

Static typestate analyses [53, 115, 82] can help capture many semantic or behavioral bugs in a program early in the development cycle. Although efficient, various complex features of real-world programs make static typestate analysis a hard problem. Most notorious of these features are (1) *aliasing* in a language with dynamic memory support, and (2) complex control and data flow in a program. *Aliasing* makes it hard to precisely track a typestate transformation over a data with various active aliases, while complex control flow makes it hard to choose the precise transition of a typestate automata.

Dynamic analysis approaches, like *runtime monitoring* [74] overcome these challenges by reducing typestate property checking to runtime checks. For example, the state of a File in the FileManager example above may be explicitly checked via a non-null value of a field and

updating this field appropriately in *close* and *open* methods. Although easier to program, these runtime checks are extremely costly for a non-trivial typestate property over a rich program. Inefficient checking can cause both runtime penalties and further make the program hard to be comprehended and debugged. Many of the typestate analysis tools and frameworks take a *hybrid* approach for typestate analysis. These works are discussed in detail in the chapter on *Related Work*(Chapter 3).

Most typestate analysis approaches enforce typestate properties via- (1) a static/dynamic program analysis over the program [53, 20], or (2) a disciplined use of member variables and fields. These approaches have been efficiently used to codify and check many sophisticated state-dependent properties of object-oriented programs. It has been used, for instance, to verify object invariants in .NET [82], to verify that Java programs adhere to object protocols [53, 18, 20], and to check that groups of objects collaborate with each other according to an interaction specification [95, 66], etc. There is another line of work in typestate analyses which integrates typestates in programming languages as first class members. These languages are called *typestate-oriented programming languages* [9, 31]. They allow expressing typestate abstractions and properties directly in the programs which may then be enforced statically or dynamically.

1.3 Objectives of this Research

The problem of typestate analysis of a property over a given program is both useful and challenging as argued in the earlier sections. These challenges arise either due to the richness of the property being analyzed or due to the complexity of the input program. Figure 1.2 plots these challenges along two dimensions. The x-axis shows an increasing complexity of property (ϕ) being analyzed, while the y-axis shows an increasing complexity of program (P) being analyzed for ϕ. The original typestate work due to Strom and Yemini [115] (point(1) on the plot), presented a typestate analysis for regular typestate properties over a simple programming language where all the complex features like aliases were abstracted away. Although simple to implement, this approach cannot handle real programs from an imperative language like Java or C++. A much more pragmatic typestate analysis should allow more complex programming language features like references, heap allocations and aliasing. Most of the state-of-the-art static typestate analyses belong to this category, shown by point (2) on the plot. Our objective in this thesis is to expand the envelope of typestate analyses along both these axes.

- *Along the axis of the complexity of Programs*, the first part of the thesis provides a static typestate analysis for Android applications (point(3) on the plot). These applications

are imperative programs with intricate control flow properties like, asynchronous call and callback handlers, event-handlers, Android framework enforced control and data flows, etc. A sound and practically precise static analysis over these applications require a correct model and semantics of these program features. These features are not unique to Android and are shared by many other platforms like iOS, Microsoft Windows mobile OS and programming models and frameworks like browser applications, etc.

- *Along the other axis is the richness of properties being verified.* The second part of the thesis provides a generalized notion of Presburger definable parameterized typestates, a core-language with a *dependent type system* to statically enforce these typestates (point(4) on the plot). This allows us to define many *non-regular* program properties of interest which are untamed by known typestate works.

There are scopes for improvements along both the axes. For example, one of the possible improvements is to extend the Presburger definable parameterized typestates, and the underlying core language and typestate system with complex language features like asynchrony and concurrency (point(5) on the plot). This will allow us to model *session types* [120] and related properties using parameterized typestates. Along the other direction, it might be interesting to look beyond Presburger definable typestate properties(point(6) and point(7) on the plot), thereby giving a much expressive typestate properties verification. We leave these as possible future extensions.

Besides these axes, automatic inductive typechecking of p-typestate properties and automatic verification of other rich verification systems usually requires a programmer to annotate loop invariants. Unfortunately, this is a challenging task even for an experienced programmer. To placate this burden from the programmer, we present a novel loop invariant calculation approach using loop acceleration technique for a Presburger definable transition system.

Next, we briefly introduce our contribution to each of these works along the two axes and our loop invariant calculation approach:

1.3.1 Complexity of the Programs

In practice, typestates define a valid sequence of operations on data with mutable states and pre-requisites on the state of the data for a method call to be valid. Thus, analyzing typestates [53, 82] requires a sound capture of the sequence of operations and state of data. Such flow-sensitive analyses that precisely captures the sequence of operations require correct capturing of program's control flow. Flow sensitivity is simple to achieve in sequential programs with procedures and there is a rich body of research on this [110, 28], but it is harder to achieve

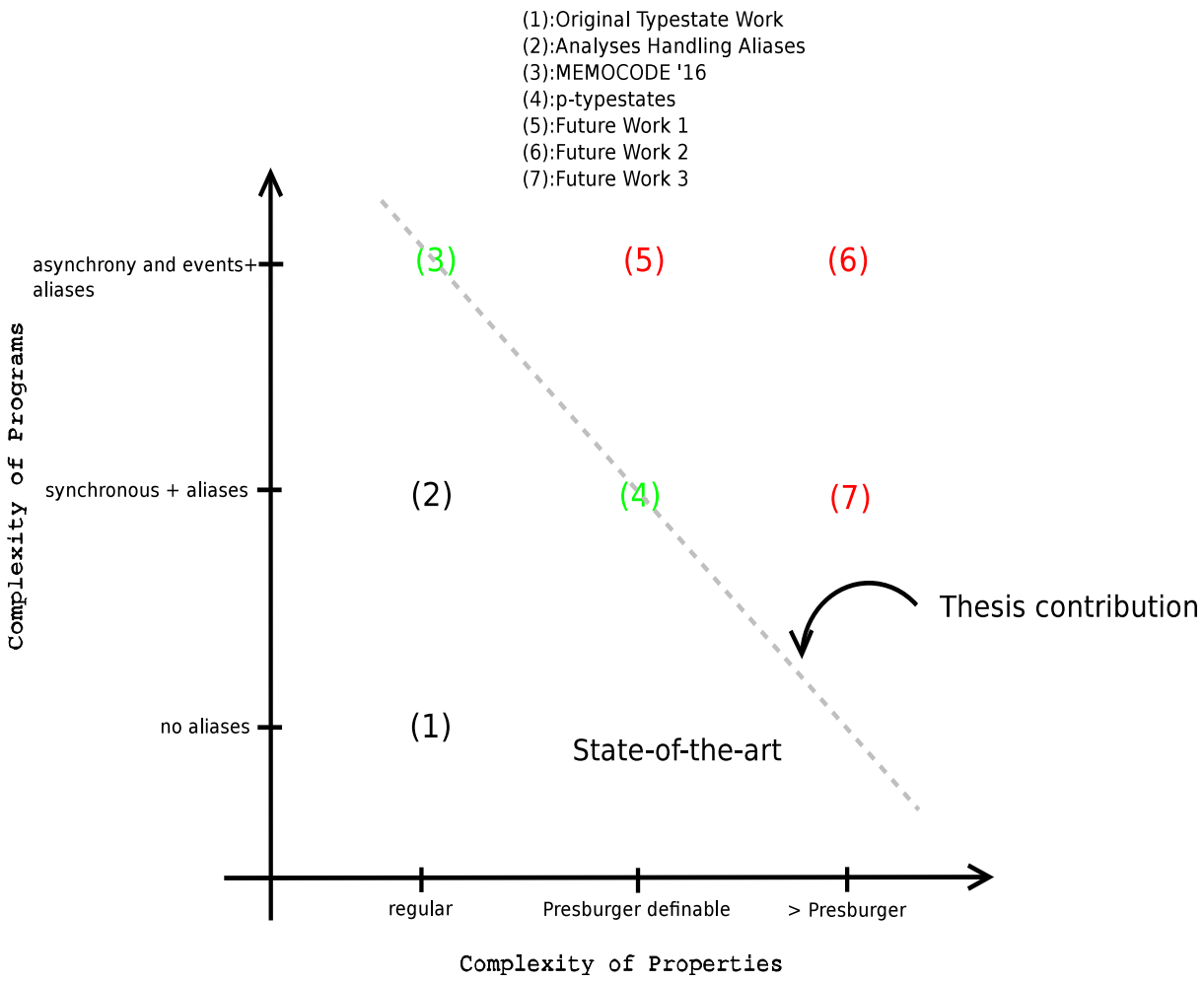


Figure 1.2: Our contributions against the challenges in typestate analyses.

for programs with richer control flow structures like asynchronous calls, callbacks, concurrency, etc. This, in turn, makes a sound typestate analysis for such programs a hard problem. Moreover, these programs and programming models are increasingly becoming popular with the rise of asynchronous, event-based programming prevalent in mobile systems like Android, iOS, etc. Further, these systems have a rich set of resources and protocols associated with them and checking the correctness of these protocols and their usage is important to find errors which otherwise might cause program failures and open security vulnerabilities in these programs. Since these protocols and their usage are classic examples of typestates, extending typestate analyses to such richer programs with complex features is an important research problem. Programs like Android applications have complex control flow semantics due to programming features like asynchrony, concurrency, distributed control flow and other Android framework induced control flow semantics. Unfortunately, most static analysis works on Android applications do

not soundly model this semantics and focus merely on extending the known static analysis techniques to Android applications. For instance, they treat asynchronous calls and communications as regular synchronous calls. These works further miss many of the semantic features of Android induced control flow. This practically, beats the purpose of static analysis approach, leaving it ineffective in proving the absence of errors. Thus the complexity of the programs being analyzed is a major challenge to efficient and precise typestate analysis.

To tackle these challenges, we present an asynchrony-aware static analysis for Android applications and introduce a program representation for Android applications which correctly models asynchrony and other framework induced control flow semantics of these programs (for single threaded non-preemptive semantics). This intermediate representation which we name *Android Inter Component Control Flow Graph (AICCFG)*, correctly models the Android asynchronous *Inter Component Communicationn* and other control flow semantics for single threaded Android applications with non-preemptive method callbacks. We also present this semantics formally and discuss how AICCFG captures this semantics. This aids in developing an asynchrony-aware static analysis over such single threaded Android applications. Android applications use a rich set of resources such as camera and media player whose safe usage is governed by some state machines. Besides there is an extensive use of other APIs like APIs for security, communication, etc., each with its associated resource usage property. To reason about these properties we build a typestate analysis over the intermediate program representation and compare it against other asynchrony-unaware static analyses for Android. We empirically show the effectiveness of our approach to both reducing false positives and capturing typestate violations which are missed by other un-sound approaches. The details of the work are discussed in Chapter 4

1.3.2 Complexity of the Properties

Classical typestates could be supported or integrated into programming languages just like types. Typestates are a part of many research and industrial programming languages. This allows one to model and verify systems for many useful program properties and capture these typestate violations during the program development phase, obviating the need for costly and error-prone runtime checks. However, classical typestates can only model properties over a regular language domain. For instance, using these we can verify a property such as- "pop should be called on a stack only after a push operation", but we cannot verify a non-regular program property such as- " the number of push operations should be at least equal to the number of pop operations". There are many useful control flow and communication properties which are beyond the domain of regular languages. For instance, "a well-formed XML file must

have a matching number of opening and closing XML tags" or "a session between a sender process and a receiver process must the have the number of items sent as greater than or equal to the number of items received.

We aim at overcoming this expressive limitation of classical typestates by defining a generalized notion of typestate, called parameterized typestates (p-typestates) as a programming language feature (a type system) which is expressive enough to model many non-regular program properties and yet has a decidable and efficient decision procedure for type-checking. This includes, formalizing the concept of p-typestate and related properties. We present a property automaton called as *p-typestate property automaton* which defines the properties verifiable using p-typestates.

Presburegr definable p-typestates allow a programmer to model a non-regular program property. Analogous to a typestate property automaton used to define a regular typestate property, we define a *p-typestate property automaton*. A p-typestate property automaton is a *multiple counter system*. The states of a p-typestate property automaton are same as a regular typestate property automaton, with a set of Presburegr definable formulas over set of auxiliary counter variables, associated with each state. The transitions are defined over a state, a Presburger formula and an input string defined by some program operation. The details of a p-typestate property automaton are discussed in Chapter 2.

The state-of-the-art programming language based typestate verification works [9, 82] use simple types to encode typestates and typestate properties, which are then typechecked either statically or dynamically. This is generally done by using simple types to capture a state of a typestate property automaton and annotating each method of the program with pre- and post- typestate annotations. This allows a Hoare style program verification for a given typestate property. In a similar way, we too provide language support for modeling a p-typestate property, we present a typestate-oriented programming language where a programmer can encode a p-typestate and a p-typestate property as pre- and post- p-typestate annotations. Unfortunately, simple types lack expressiveness to encode a p-typestate state, i.e. a state dependent on a Presburger formula over integer variables. Thus, we need richer type theories for p-typestates. We implement the concept using dependent type theory, modeling p-typestate as dependent types parameterized over Presburger definable formulas and a set of regular typestate states. Restricting the dependent types over Presburger formulas provides sufficient expressiveness to the type system to capture p-typestate properties (given by some p-typestate automaton) and decidable properties of Presburger logical family yields a decidable type-checking for our type system. We also present a core dependently typed programming language with the p-typestate type system. We further provide correctness (soundness of the type system) and decidability

guarantees of the formalization and the p-typestate system. Finally, we model and verify several practical non-regular program properties using our language and p-typestates.

1.3.3 Loop Invariant Calculation for Presburger Definable Systems

For each of these axes, an automatic inductive type checking or verification of a rich program property is hindered by the calculation of invariants for loops and recursive data structures. For instance various refinement types works [112, 126] using rich type theories to verify invariants over data require loop invariants or invariants over recursive data to be annotated by the programmer. Since the Presburger definable p-typestate type system also uses dependent types, we too require the programmer to provide the loop invariants. These invariants can then be composed with the incoming Presburger logical formulas to generate an inductive proof for the p-typestate property of interest. We provide a novel loop acceleration based loop invariant calculation technique for Presburger definable systems in our work which placates this burden from the programmer for the class of properties captured via our p-typestates. The details of the approach are presented in Chapter 5.

1.4 Our Contributions

- We present a sound model of asynchronous control flow and component lifecycle semantics (single threaded, non-preemptive semantics) in Android application. Built on this model, we present an intermediate program representation of Android applications called *Android Inter-component Control Flow Graph* (AICCFG). This representation aides in designing sound and precise static analyses for Android application which has been ignored by the state-of-the-art static analyses works for these applications.

- We develop an asynchrony-aware static typestate analysis over the AICCFG for tracking Android resource APIs violations for a variety of Android resources like Camera, FileManager, MediaPlayer, etc. The approach is a generic asynchrony-aware static analysis for Android applications and can be used to develop other static analyses.

- We present a set of benchmark applications (single threaded applications) called as Asyncbench, comprising of API resource usages for a set of resource type whose precise typestate analysis requires sound modeling of Android control flow semantics (single threaded, non-preemptive semantics). We analyze these benchmark applications using our analysis and compare the results against a popular asynchrony-unaware static analysis approach [76] and demonstrate the efficacy of AICCFG and the asynchrony-aware typestate analysis.

- We introduce the concept of Parameterized typestates and define Presburger definable parameterized typestates and present a dependent type system to capture and enforce parameterized typestate properties and define and implement a core typestate oriented language with static p-typestate typechecking.
- We present a formal study and proof of Soundness and decidability of typechecking for the dependent type system implementing p-typestates.
- We further show the effectiveness of the p-typestate by implementing many real-world programs enforcing p-typestate properties which cannot be specified using regular typestates and other typestate oriented programming languages.
- We present a novel loop invariant calculation for core language programs specifying p-typestate properties.

1.5 Organization of the Thesis

This thesis is organized as follows. In Chapter 2, we present a brief introduction to some of the important preliminary and background material which required to easily understand the technical contributions and ideas discussed in the rest of the thesis. We begin with a brief background of Android applications, static analysis in general and particularly in the context of an Android application. This is followed by brief introduction to the background on *types*, *type systems* and richer type theories like *dependent types*. We also discuss briefly mathematical logic and logical families like Presburegr arithmetic. This is followed by a small discussion about the verification of infinite-state systems. These background topics will help to understand the ideas discussed in Chapter 4 and Chapter 5.

In Chapter 3, we present a detailed survey of existing works related to Android formal modeling, asynchronous static analysis, static analysis for Android applications, existing typestate analysis for Android and other programs, etc. This is followed by mentioning of existing related works with typestate-oriented programming, dependently typed languages, dependent types extension of simply typed languages, etc. Finally, we present existing works from the domain of automatic loop invariant calculation.

In Chapter 4, we motivate for a sound model of Android asynchronous control flow, following which we provide such a formal model and its semantics. We present an intermediate program representation for Android applications, which captures our formal model and semantics. We motivate the need for an asynchrony-aware static analysis for Android applications and present an asynchrony-aware, static typestate analysis over our intermediate program representation.

We empirically, show the effectiveness of our model and analysis and compare it against other state-of-the-art analyses.

In Chapter 5, we begin by defining regular typestates and the motivation for a more generalized notion of typestates. Following this, we introduce the idea of Parameterized typestates (p-typestates) and formalize the concept. We present a way to implementing these p-typestates in a typestate-oriented programming language using dependent types. We discuss the challenges associated with the p-typestate type system, typechecking and our approach to handling these challenges. We conclude the chapter by presenting a novel and simple loop invariant calculation technique and its integration to our language. We empirically show the use cases for our language and invariant calculation.

Finally, we summarize our work in Chapter 6 and discuss some of the key directions in which this work can be extended.

Chapter 2

Preliminaries

Our thesis covers works, contributions and ideas from various domains like Android applications, program analysis, formal modeling, type theory, etc. This chapter provides a brief introduction to some of these topics. This is not a complete description of these works, as that will be out of the scope of this thesis. We provide a set of references for further reading where ever required.

2.1 Introduction to Android

Chapter 4 discusses an asynchrony-aware static analysis of programs with complex programming features like *asynchronous control flow* and other *framework* or system induced control flow semantics. We use Android applications as a target use case for the approach. This section provides the basic background on Android applications and control flow in Android, which will aid in easy comprehension of the topics discussed in Chapter 4.

2.1.1 Android Architecture

Android is the fastest growing and heaviest used mobile operating system [3]. The Android platform consists of the core operating system and a stack of software components. Android applications (apps in short) are mostly written in Java, however, newer versions of Android also support applications written in Kotlin or C++. The Android SDK [7] tool compiles your code along with any *data* and *resource* files into an APK, an *Android package*, which is an *archive* file with a .apk suffix. An APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app. An Android application sits on a device and runs by interacting with the user, the *Android Application Framework*, Android system processes and other layers of the *Android Platform*.

Figure 2.1 presents a layered architecture of the Android operating system kernel and other layers of the software stack. The lowest layer and the foundation of the Android platform

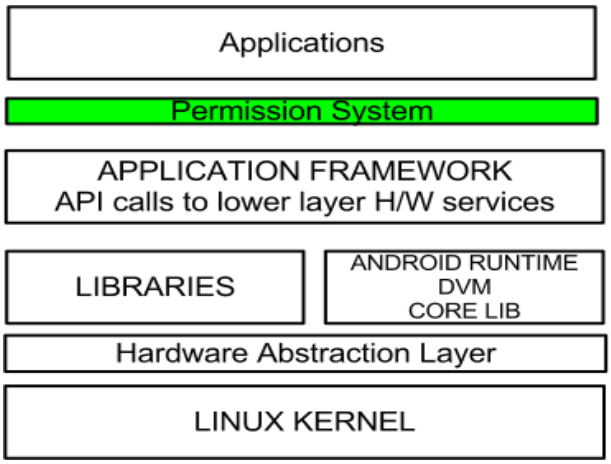


Figure 2.1: The Android Software Stack

is the *Linux kernel*. Many of the higher layers rely on the kernel, for instance, the *Android Runtime* relies on the kernel for threading and low-level memory management. The next layer is the *Hardware Abstraction Layer*(HAL) which provides a standard interface to expose device hardware capabilities to the higher levels of the stack, such as the application framework. The HAL consists of multiple library modules one for each type of hardware component like *camera* or *bluetooth*, etc. The next layer has two components , the Android Runtime (ART) and the Native Libraries. The ART is written to run multiple virtual machines (each application runs in a separate process in a separate VM) on low memory devices by executing *Dalvik executables* (DEX) [2] files. The core Android system components like the HAL and the ART are written in C/C++ and require native library support written in C/C++. These form the Native library of the stack.

The next important layer from the applications behavior perspective, is the Android Application Framework, a Java API framework. These APIs expose the entire feature set of the Android OS. The framework also provides APIs to expose the functionalities of some of the Native libraries to the apps. The Java API framework has a rich *View System* to develop rich UI for the apps. It contains a *Resource Manager* to access non-code resources like the localized strings, graphics and layout file, etc. and these resources are packaged alongside the application code in the APK. The Android *Activity Manager* API, manages the *lifecycle* of the application and application *Components* and manages other control flows in the application. Besides these and other system services, the APIs also contain *Content Providers* which act as databases for the applications. At the top of the stack sits the Android application which interacts with the lower layers through the Application Framework. These interactions are protected via a capability-based *Android Permission System*.

2.1.2 Android Applications

Android restricts each application to its own *security sandbox*. Each application behaves as a separate user process in the Android operating system, which is a multi-user Linux system and is assigned a unique Linux ID. These IDs are used to implement the Android's main security feature, the *Android permission system*. Each application runs in its own process in a separate virtual machine and runs in isolation from other apps. Next, we define the structure of an Android application, the permission system and the Android resources, framework and other Android application features. We present a running example application showing each of these features.

2.1.2.1 Android Application Components

An Android application is composed of four kinds of *components*. An application has a set of functionalities like showing the interface to the user, sending or receiving broadcasts, providing database functionalities, etc. Android provides a modular design to implement these functionalities in an application. An application has four different types of components, see Figure 2.2. Each component in Figure 2.2 is annotated with an actual application image, closely resembling the component type and the arrows represent the possible interactions between the components. The (incoming) *start* arrows to the three component types (other than *Content Providers*) depict that each of these component types can be an entry to the application.

- **Activity**: Activities are the most fundamental components of an application and form a major percentage of an application's source code. Activities form the User Interface of the application and serve as the entry point for a user's interaction with the application. Informally, any UI component or screen of an application with which a user is interacting is some variant of Activity. Practically an Activity is implemented by extending the base Activity class of the Application Framework. Most apps contain multiple screens and thus comprise of multiple Activities. Typically one of the Activities of the application is specified (in the *Manifest*) as the *main* Activity, which is the first screen which pops up when the user launches the application. Each Activity (and similarly other components) are a bunch of event callback handling methods like *onCreate, onStart* etc. which are invoked by the Application Framework (particularly, the Activity Manager) based on certain user and system events and the lifecycle of the Activity(and other components). Listing 2.1 shows a code fragment for an Activity component named MyActivity. This toy example activity simply creates the UI on the screen via a call setContentView(R.layout.activity_main) in the onCreate callback. The setContentView takes a resource layout file as input and

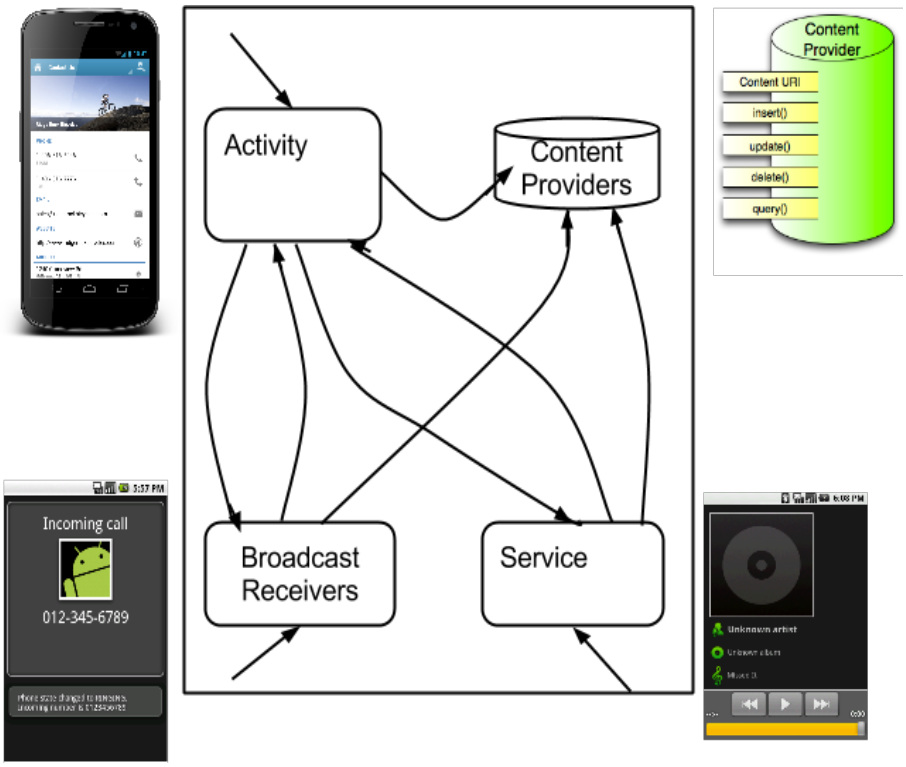


Figure 2.2: Android Application Components

draws the layout instantiating required *Android Views*. The activity displays a message on the screen showing the current callback being executed. The features in MyActivity interact with other components of the application and will be explained gradually, as other components are explained.

Listing 2.1: Example : MyActivity

```
1   package thesis.example.android;
2
3   import android.os.Bundle;
4   import android.app.Activity;
5   import android.util.Log;
6
7   public class MyActivity extends Activity{
8
9   // A string message used to track Activity lifecycle
10  String welcomeString = "Exmaple Application";
11
12
13  /** Called when the activity is first created. */
14  @Override
15  public void onCreate(Bundle savedInstanceState) {
16  super.onCreate(savedInstanceState);
17  setContentView(R.layout.activity_main);
```

```
18          Log.d(welcomeString, "onCreate() finished");
19          }
20
21          /** method to add the name and salary of an employee to the MyContentProvider content
                provider**/
22           public void onClickAddName(View view) {
23                   // Add a new Employee record
24                   ContentValues values = new ContentValues();
25                   values.put(MyContentProvider.e_Name,
26           ((EditText)findViewById(R.id.editText2)).getText().toString());
27
28           values.put(StudentsProvider.e_Salary,
29           ((EditText)findViewById(R.id.editText3)).getText().toString());
30
31           Uri uri = getContentResolver().insert(
32           MyContentProvider.CONTENT_URI, values);
33
34           Toast.makeText(getBaseContext(),
35           uri.toString(), Toast.LENGTH_LONG).show();
36           }
37
38           /** methods to start and stop the MyService service **/
39           public void startService(View view) {
40                   startService(new Intent(getBaseContext(), MyService.class));
41           }
42
43           // Method to stop the service
44           public void stopService(View view) {
45                   stopService(new Intent(getBaseContext(), MyService.class));
46           }
47
48          /** Called when the activity is about to become visible. */
49          @Override
50          protected void onStart() {
51          super.onStart();
52          startService(this);
53          Log.d(welcomeString, "onStart() finished");
54          }
55
56          /** Called when the activity has become visible. */
57          @Override
58          protected void onResume() {
59          super.onResume();
60          Log.d(welcomeString, "onResume() finished");
61          }
62
63          /** Called when another activity is taking focus. */
64          @Override
65          protected void onPause() {
66          super.onPause();
67          Log.d(welcomeString, "onPause() finished");
68          }
69
```

```
70          /** Called when the activity is no longer visible. */
71          @Override
72          protected void onStop() {
73          super.onStop();
74          Log.d(welcomeString, "onStop finished");
75          }
76
77          /** Called just before the activity is destroyed. */
78          @Override
79          public void onDestroy() {
80          super.onDestroy();
81          Log.d(welcomeString, "onDestroy() finished");
82          }
83  }
```

- **Service**: Many applications have some long-running operations in the background and do not provide a user interface. These operations will affect the responsiveness of the UI if run in an Activity. A Service is an application component which includes such operations. Another application component or a system process can start a Service and it continues to run in the background even if the user switches to another application. Additionally, Services also allow other components to *bind* to them to interact and perform interprocess communication (IPC). For example, a service can manage network transactions or run a music app in the background. Listing 2.2 shows an example Android service, which can be either *bounded* (provides an onBind() method) or can be *started* (provides an onStartCommand()) method. A started service once started, runs indefinitely, while a bounded service provides a *client-server* interface that allows other components to interact with this service, send request, get response, etc., via Inter Component Communication (ICC) across components of the same process or application or even across processes. MyActivity starts this example service using a startService method call at line number 52 (in Listing 2.1).

Listing 2.2: Example : MyService

```
1   package thesis.example.android;
2
3   import android.app.Service;
4   import android.content.Intent;
5   import android.os.IBinder;
6   import android.widget.Toast;
7
8   public class MyService extends Service {
9
10          public IBinder serviceBinder;
11                  @Override
12          public void onCreate() {
```

```
13          // TODO Auto-generated method stub
14          super.onCreate();
15          }
16
17          @Override
18          public IBinder onBind(Intent intent) {
19          // TODO Auto-generated method stub
20          return serviceBinder;
21          }
22
23          @Override
24          public int onStartCommand(Intent intent, int flags, int startId) {
25          // TODO Auto-generated method stub
26          super.onStartCommand(intent, flags, startId);
27          Toast.makeText(this, "My Service Started", Toast.LENGTH_LONG).show();
28          return START_STICKY;
29          }
30
31          @Override
32          public void onDestroy() {
33          // TODO Auto-generated method stub
34          super.onDestroy();
35          Toast.makeText(this, "My Service Destroyed", Toast.LENGTH_LONG).show();
36          }
37
38          @Override
39          public boolean onUnbind(Intent intent) {
40          // TODO Auto-generated method stub
41          return super.onUnbind(intent);
42          }
43
44
45  }
```

- **Content Provider**: An Android application might need to store and manage some data for itself or it might need to share some persistent data with other applications. Content Providers can help an application manage access to data stored by it, by other apps and provide a way to share this data with other apps. Thus Content Providers are private databases of apps. An Android Content Provider is implemented by extending the Android framework's ContentProvider class and must implement required a set of APIs allowing other applications to perform transactions. For instance, an e-mail client application needs to store the list of contacts or the incoming mails list, which it might need to share with another application. A particular advantage of Content Providers is that they allow granular control over the permissions for accessing data. The application can grant a blanket permission to access data or have a more fine-grained access to READ or WRITE data.

Listing 2.3: Example : MyContentProvider

```
1  package thesis.example.android;
2
3  import java.sql.SQLException;
4
5  import android.content.ContentProvider;
6  import android.content.ContentUris;
7  import android.content.ContentValues;
8  import android.content.Context;
9  import android.database.Cursor;
10 import android.database.sqlite.SQLiteDatabase;
11 import android.database.sqlite.SQLiteDatabase.CursorFactory;
12 import android.database.sqlite.SQLiteOpenHelper;
13 import android.net.Uri;
14
15 // Example content provider with an employee database for a company
16 public class MyContentProvider extends ContentProvider{
17
18         static final String NAME = "thesis.example.android.myapplication";
19         static final String URL = "content://"+NAME+"myContentProvider";
20         static final Uri uri = Uri.parse(URL);
21
22         static final String e_Name = "name";
23         static final String e_Id = "id";
24         static final String e_Salary = "salary";
25
26
27         private SQLiteDatabase sqldb;
28         static final String DB_NAME = "Company";
29         static final String EMPLOYEE_TABLE_NAME = "employee";
30         static final int version = 1;
31         // prepare a create table query string
32         static final String CREATE_COM_TABLE =
33                 " CREATE TABLE "+ EMPLOYEE_TABLE_NAME +
34                 " ( e_Id INTEGER PRIMARY KEY, " +
35                 " e_Name TEXT NOT NULL, " +
36                 " e_Salaray DOUBLE NOT NULL);";
37
38         private static class DataBaseHelper extends SQLiteOpenHelper{
39
40         public DataBaseHelper(Context context) {
41                 super(context, DB_NAME, null, version);
42         // TODO Auto-generated constructor stub
43         }
44
45         @Override
46         public void onCreate(SQLiteDatabase db) {
47         // TODO Auto-generated method stub
48         db.execSQL(CREATE_COM_TABLE);
49
50         }
51
```

```
52          @Override
53          public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
54          // TODO Auto-generated method stub
55          db.execSQL("DROP TABLE IF EXISTS "+ EMPLOYEE_TABLE_NAME);
56          onCreate(db);
57
58          }
59
60          }
61
62          @Override
63          public boolean onCreate() {
64          // TODO Auto-generated method stub
65                  Context context = getContext();
66                  DataBaseHelper dbHelper = new DataBaseHelper(context);
67
68                  return dbHelper.getWritableDatabase() == null ?false : true;
69          }
70
71          @Override
72          public Cursor query(Uri uri, String[] projection, String selection,
73                  String[] selectionArgs, String sortOrder) {
74                  // TODO Auto-generated method stub
75                  return null;
76          }
77
78          @Override
79          public String getType(Uri uri) {
80          // TODO Auto-generated method stub
81
82          }
83
84          @Override
85          public Uri insert(Uri uri, ContentValues values) {
86                  // TODO Auto-generated method stub
87                  long row = sqldb.insert(EMPLOYEE_TABLE_NAME, "", values);
88                  if(row > 0){
89                          Uri row_Uri = ContentUris.withAppendedId(uri, row);
90                          getContext().getContentResolver().notifyChange(row_Uri, null);
91                          return row_Uri;
92                  }else
93                          return null;
94          }
95
96          @Override
97          public int delete(Uri uri, String selection, String[] selectionArgs) {
98                  // TODO Auto-generated method stub
99                  return 0;
100         }
101
102         @Override
103         public int update(Uri uri, ContentValues values, String selection,
104                 String[] selectionArgs) {
```

```
105                 // TODO Auto-generated method stub
106                 return 0;
107         }
108
109 }
```

Listing 2.3 shows a Content Provider (MyContentProvider) from the running example application. The content provider contains a *SQLiteDatabase* instance and a DatabaseHelper inner class in the MyContentProvider class. The content provider is required to implement methods from the ContentProvider class for standard create, update, delete and insert database operations.

- **Broadcast Receiver**: An Android application needs to send and receive broadcast messages to/from Android system and other applications. For instance, a music player app must receive and pause the playing music on an incoming call. Broadcast Receivers are the components of an application for sending and receiving such broadcasts. An application can receive broadcasts against an event (user or system) by defining a receiver component and registering it for particular kind of events or *Intents*. A broadcast receiver can be registered in two ways, through manifest declaration or through context registration. If a broadcast receiver is declared in the application's manifest, the system forwards the message (launches the app if not already running) to the application.

  Listing 2.4 shows a Broadcast Receiver with a lifecycle method onReceive() method. The MyActivity sends a broadcast intent which is intercepted by MyBroadcastReceiver. The receiver registers for the broadcast intent in the *AndroidManifest.xml* file (explained later).

Listing 2.4: Example : MyBroadcastReceiver

```
1         package thesis.example.android;
2         import android.content.BroadcastReceiver;
3         import android.content.Context;
4         import android.content.Intent;
5         import android.util.Log;
6
7         public class MyReceiver extends BroadcastReceiver {
8                 public static final String msg = "My receiver";
9                 @Overrid
10                public void onReceive(Context context, Intent intent) {
11                        // TODO Auto-generated method stub
12                        Log.d(msg, "received the msg");
13                }
14
15        }
```

Each of these component types has a set of APIs for fine-grained control and usage. Details of these component types can be found from the official Android API guide [7].

2.1.2.2 Android resources and Manifest

Each Android application is compiled and packaged as *.apk* package. The package contains the *dalvik* bytecode [2] for the application along with a set of resources required by the Android framework for installing and executing the application.

- **Application Resources**: Each application uses resources like images, strings, etc., which are used by certain components of the application or are necessary to install and execute the application in the *application framework*. These resources include, XML files needed by the application like the layout XML files (except the *manifest.xml*), string and other constant values used in the application, images, etc.

- **Application Manifest** : Each Android application is mandatorily packaged with an *AndroidManifest.xml* file which is a file containing useful information about the application. For instance, the Manifest contains information regarding each component of the application along with their properties, the Java package for the application, the *permissions* required by the application, libraries required for the execution of the application, etc. The structure of the Manifest is like a typical XML file with appropriate sets of elements and corresponding properties. Figure 2.3 presents an example code fragment showing a part of an AndroidManifest.xml file for the running example application with a single activity named MyActivity, a service, a receiver and a content provider. The Manifest first defines a user-defined permission and then makes the application ask for this permission. Android has an *install-time* static permission system such that an application must ask for all the permissions upfront in its Manifest before the application is installed and these must be granted by the user for a successful installation of the application.

  The Manifest also defines or affects certain crucial aspects of the control flow of an application, as the Android System reads the Manifest to define: the *main* activity of the application (a kind of entry point to the application when it is started), the set of public components of the application, defining the secondary entry points of the application which can be invoked by other applications through asynchronous message passing mechanism in Android called *Inter-Component Communication* (ICC). ICC includes a set of *Intent filters*, defining on what *data* or *action*, the application can be invoked, etc. Thus in summary, the Android Manifest associated with an application controls or rather affects almost all major stages of the application's execution from its creation, its interaction

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.thesis.example.android"
package="thesis.example.android.myapplication">
    <permission android:name="android.example.iisc.MY_PERMISSION" . . . />
    <uses-permission android:name="android.example.iisc.MY_PERMISSION" />
    . . .
    <application . . .>
        <activity android:name="android.example.iisc.MyActivity"
                  android:permission="android.example.iisc.MY_PERMISSION"
                  . . . >
                   <intent-filter>
                   <action android:name="android.intent.action.MAIN" />

                   <category android:name="android.intent.category.LAUNCHER" />
                   </intent-filter>
            . . .
        </activity>

        <receiver android:name="MyReceiver">
                <intent-filter>
                <action android:name="com.tutorialspoint.CUSTOM_INTENT">
                </action>
        </intent-filter>

        </receiver>
    </application>


    <service android:name="MyService" />

    <provider android:name="MyContentProvider"
    android:authorities="thesis.example.android.myapplication.MyContentProvider"/>
    </application>

  </manifest>
```

Figure 2.3: Example : Application Manifest Fragment

with other components and applications, its access to useful system resources via permission, its dependence on other libraries, etc. making it the most fundamental component of the application.

2.1.3 Control Flow in Android Applications

Android applications are generally written in Java, but the control and data flow in an application differ substantially from that of a typical Java program. An Android application interacts with the Android application framework and Android System processes during execution. There is no concept of a single entry method like *main* in a typical Java application. Contrary to this, an application has a set of *public* components (a component is tagged public in the associated *AndroidManifest.xml file*) which act as possible entry points to the application on certain user or system events. The control flow in each component is governed by its interaction with the user, other components and the Android system. The Android systems further define a *lifecycle* for each type of component governing how the control flows between different methods of a component.

2.1.3.1 Android Activity Lifecycle

An Android activity contains a bunch of framework declared lifecycle methods like *onCreate, onStart, onDestroy, etc.* The activity life cycle can be defined as a binary relationship between these methods such that two methods are related if the second can be executed after the first. This possible execution ordering is managed by the Android Application Framework (Activity Manager) and depends on the *state* of the Activity. Figure 2.4 is adapted from the Android official page and depicts the Activity lifecycle visually. An activity during its lifetime is in four main states *viz.*, *started, running, paused* and *killed*. Each lifecycle callback method is invoked and runs without preemption and defines transition of activity between these states. The component lifecycle associated with each component type aids in providing an optimum user experience and system resource usage. An activity starts and the system creates or draws its user interface on the screen (declared in an XML layout file associated with the application) in its *onCreate* method. The onCreate method also performs other fundamental setup tasks for an activity like defining member variables and configuring other parameters and UI of activity. Once the activity is created its *onStart* and *onResume* methods are called before any switch to a new state.

The onStart method call makes the activity visible to the user while the system prepares the activity to come to the foreground and become interactive. The finish of onStart makes the Android system invoke the onResume method when the activity really comes to the foreground.

This switches the activity to the *running* state. The activity stays in this state until some user or system event takes the focus away from the current activity, like hitting the *back* button or starting another activity or application. These events trigger the paused state of the activity by making the system invoke the *onPause* method. This is the first indication that the user is leaving the current activity and hence this method performs routine tasks related to saving the current state of the activity to which it can again be resumed. onPause takes the current activity to the background which can then either be restored to the foreground by the Android system by calling onResume again or can be pushed to an invisible state by the system call to *onStop*.

In the onStop method, the activity should release almost all the resources which are no longer needed for the activity. For instance, a Broadcast receiver registered earlier may be unregistered in onStop or temporary data should be saved to permanent storage. From the stopped state, the activity either comes back to start state again or finishes its task and goes away to the killed state. If the activity comes back, the system invokes *onRestart*, while if it goes away the system invokes *onDestroy* method. *onDestroy* is called before the activity is destroyed. The system invokes this method either because the activity is finishing or some other component called the *finish* method. Another reason this method can be called is when the system has resource crunch (like space) and it decides to kill the activity prematurely.

Listing 2.1 shows a typical lifecycle method for an activity. The code fragment is commented showing when a particular lifecycle callback is invoked. For simplicity, these methods just log the current callback method being executed along with the welcome message. All the lifecycle callback methods of an activity (or another component) are called asynchronously by the Android system and follow a non-preemptive semantics. Lifecycles interact with other synchronous and asynchronous calls and callbacks to create intricate control flows in Android applications.

2.1.3.2 Android Service lifecycle

Similar to Activities, Services have defined life cycles. As mentioned earlier unlike Activities, there are two ways to interact with a Service, either via normal Inter-Component Communication (ICC) through *Intents* or by *binding* to a Service. This gives two different lifecycles for these two kinds of Services. The details about the Service life cycle and life cycles associated with other components can be found on the Android official page [7].

Listing 2.2, shows some of the lifecycle callbacks like onStartCommant and onBind. The onBind method returns an *IBiunder* object to the caller (the MyActivity activity here), which can be used for sending client-server like communication requests to this Service. The onStartCom-

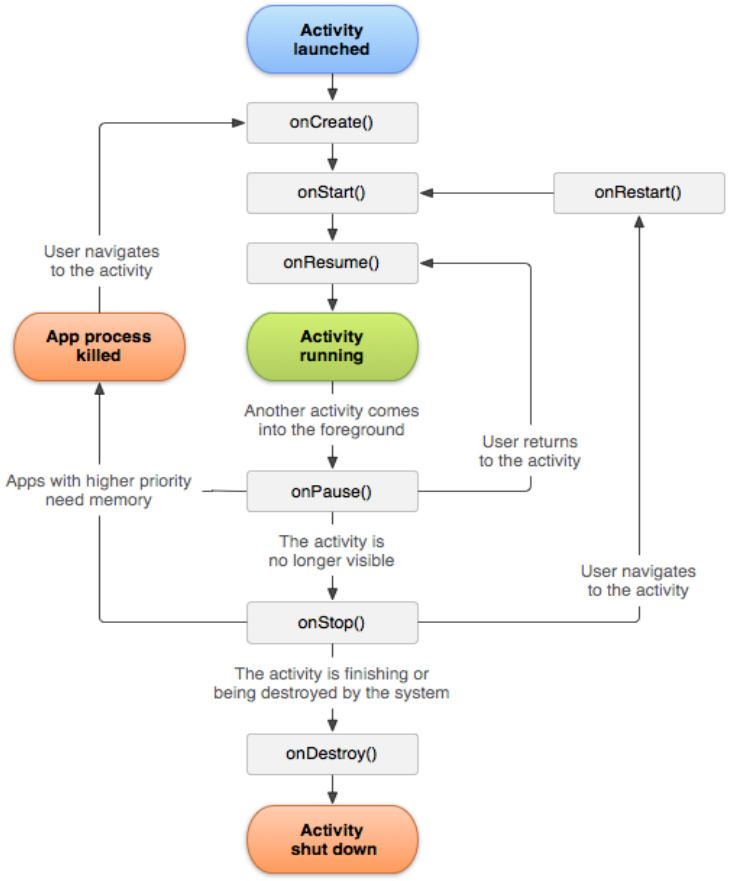


Figure 2.4: Activity Life cycle

mand method is called when the service is started to execute some task independently of the caller. Other lifecycle callback methods like onDestroy and onUnbind are called by the Android system on corresponding events.

2.1.3.3 Android ICC

The components of an Android application need to communicate (via control flow) and share data amongst each other. For instance, an *e-mail* client Activity displaying the list of emails (a display activity) in the inbox needs to start another activity to view (view activity) a selected mail. Along with the control flow, the display activity must also send some data regarding the mail being selected for the view activity. This is an example of intra application ICC. ICC is also needed across applications, For instance, the same e-mail client might need to open a PDF document received as an attachment. To achieve this, the e-mail application needs to send an ICC to the PDF-viewer application along with required data.

These inter-component communications in Android happen through *asynchronous message passing*. These asynchronous messages are called *Intents*. The activity MyActivity shown in Listing 2.1 shows how the activity interacts with a service MyService using *Intent* (line number 52 in the onStart lifecycle method), which in turn makes a call to startService() api with an Intent parameter containing explicit target *MyService.class*. Next we define *Intent* for formally.

Intent : According to the Android official page, "An Intent is an abstract description of an operation to be performed". An intent can be used to start an Activity (using *startActivity(Intent)*) or Service (startService(Intent)) or sending broadcasts (sendBroadcast(Intent...)) or binding to a Service (bindService(Intent)) or some other interaction between components.

Typically an Intent is called as follows :

```
1 Intent myIntent = new Intent (action, data);
2 startActivity(myIntent);
```

Where action defines a system or user-defined method which the target component will execute and data defines the data which may be required to complete the action. Some of the examples for action/data pair can be, ($ACTION\_DIAL/tel1234555$), this action/data pair will display the phone dialer with the tel number filled. This is an example of a system defined action. The code fragment is an example of an *implicit* intent, where the target of is dynamically chosen based on the action and data field. The components register for certain action or data values via *intent-filters* in the *AndroidManifest.xml* file and are invoked when an Intent with the corresponding action or data is sent out. Another way of sending intents is via explicit target information.

```
1 Intent myIntent = new Intent (MyTargetActivity.class, data);
2 startActivity(myIntent);
```

Here the target activity class is explicitly defined. In addition to the primary action/data attributes of Intents, there can be various *secondary attributes* attached to an Intent, like *Type* (defining the MIME data type explicitly), *Extras* (Bundle of extra information), etc.

2.1.3.4 Android Resource Protocols

Android devices have a rich set of hardware resources which are exposed via a group of the Android framework provided APIs. This allows the developers to easily use resources like *Camera, MediaPlayer, SQLiteDatabases, Files, etc.* Each of these resources has an associated *resource usage protocol* which must be respected by the developers. These usage protocols are basically typestate properties, defining the validity of a method in a given state of the resource. Developers generally enforce these protocols via explicit runtime checks on the state of the resource object.

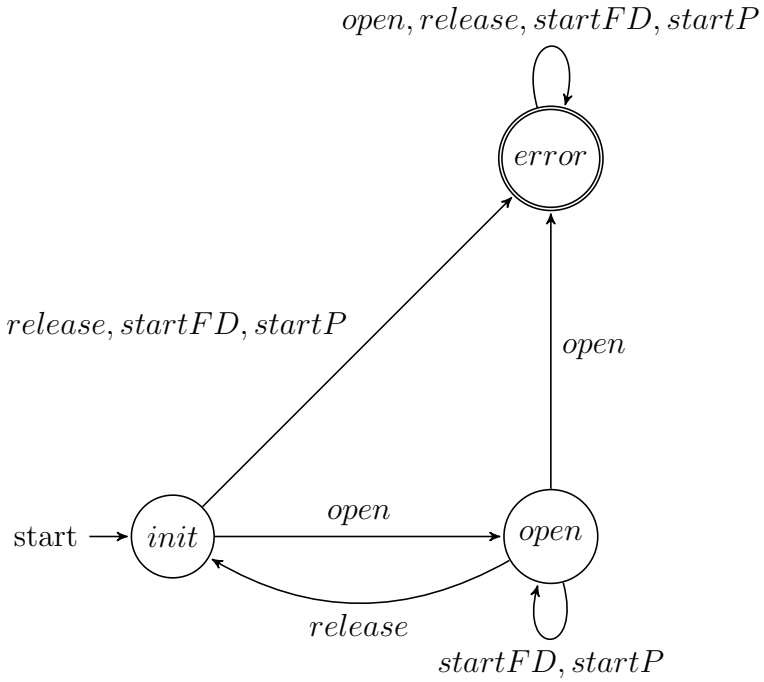


Figure 2.5: Simplified Resource Usage Protocol for Android Camera API: startFD := startFaceDetection, startP := startPreview

For example, Figure 2.5 shows the API usage protocol for the *Camera* resource. Unfortunately, enforcing these protocols via runtime checks has various limitations. Firstly, The

protocol is presented as a separate document (like a class document) which is then enforced using explicit state-checks in the program. This makes the program hard to comprehend as the program text is flooded with checks for a property which is not a part of the program logic. Secondly, these runtime checks are costly and for certain resources with complex usage protocols like *Android MediaPlayer*, the runtime cost may be substantial. Finally, this makes debugging harder, as the exceptions raised due to errors are far removed from the site of the cause of the error. Chapter 4 discusses these protocols and an analysis to statically identify these violations in detail.

2.2 Interprocedural Static Analysis

An inter-procedural data flow analysis analyzes the data flows in programs with procedures. For example, consider code fragment with a main procedure and two other procedures. Android applications like most of the real-world programs are modular and contain procedures and procedure calls. Thus, an analysis for Android applications requires inter-procedural capabilities. This can be done either statically, called as *static inter-procedural analysis* or dynamic, called as *dynamic inter-procedural analysis*. There are various well understood inter-procedural analysis approaches and underlying theories for them, like call-string based approach or *functional or iterative approach* [80]. Practically, one of the most efficient and extensively used approaches is due to Reps et al. [110] which reduces the inter-procedural data flow analysis problem to a graph reachability problem over an *exploded* inter-procedural control flow graph of the program. This approach is commonly known as the *IFDS* approach in the literature.

Android applications are highly modular and comprise of a large number of procedures or methods which are called both by the application code as well as the Android framework libraries. Thus, IFDS is extensively used in inter-procedural static analyses for Android applications [76, 125].

However, calls in Android applications may be either *synchronous*, where the caller blocks for the control to return from the callee, or *asynchronous*, where the caller continues its execution while the call is enqueued to a scheduler which dispatches the call at some later time. Furthermore, Android applications allow callbacks from the Android System in the form of asynchronous calls and other framework library callbacks. These features mandate various modifications to the original IFDS approach to incorporate asynchronous features of Android applications and its ICC. The details of these modifications are presented in Chapter 4.

2.3 Introduction to Types and Type Systems

As programs grow in size and complexity, reasoning about the behavior of these programs and debugging them becomes non-trivial and increasingly hard. This may lead to multiple programming errors which might open exploitable vulnerabilities making them prone to numerous attacks. Furthermore, there has been a striking rise in both the size and the complexity of real-world programs thus making reasoning about the behavior of the programs a major research focus of the research in the programming languages domain. Many of these works [110, 80, 39, 115, 97, 19] develop proofs and other automatic systems to debug and *verify* programs. These program verification and analysis techniques prove/verify the *correctness* of the program. In this statement *correctness* is a loaded term and needs some explanation. *Correctness* most generally means, that the " the program semantics follows the intention of the programmer". This *intention* is termed as the *verification property*. There are various well-known verification techniques and frameworks to verify a given property over a program such as, *Hoare Logic* [19], *Modal Logic* [50], *Denotational Semantics* [105], etc. These systems and frameworks have expressive power to define and prove very general (complex) program properties but are mostly manual or semi-automatic requiring significant efforts and understanding from the programmer. Another less rigorous and fully (or considerably more) automatic approach for program verification is *Types and Type systems*. In this thesis, we present a rich type system to verify rich *typestate* properties over programs. This section presents a brief introduction to the concepts of *types, type systems, richer type theories* and their use for program verification. These concepts will help in elaboration of the ideas and work presented in the Chapter on Parameterized typestates 5

2.3.1 Types and Type Systems

The main use of *type systems* is to prevent the occurrence of execution errors in a program. This informal definition of type systems requires further explanation. The simplest definition of an execution error can be the occurrence of an unexpected or unwanted software fault, like an illegal instruction fault or an illegal memory reference fault. However, execution errors might also include other unexpected behaviors of the program without any immediate visible effect. In fact, these errors are much harder to locate and debug. In general, an execution error may refer to, a program showing a semantics not intended by the programmer. We provide a formal definition of some of the useful terminologies, starting from the definition of types.

2.3.2 Typed and Untyped Languages

The *type* of a program variable defines the kind or the set of values this program variable may assume during any execution of the program. For example, if x is has a type *Boolean*, it may be assigned only values from the set { *True, False* } and not a numeric value like 3. Further, the type of a program variable also defines the set of valid operations or functions on the program variable. For instance, a function *not(x)* is a valid operation on the program variable x, with a sensible meaning, while a function *add(x,3)*, with a usual semantics of addition is not a valid operation. Programming languages where program variables may be assigned types are called *typed languages*, while those languages where program variables may assume any values during execution and place no restriction on the valid set of operations are called *untyped languages*. The simplest and the most extreme case of untyped languages is the *untyped lambda calculus* [105] which never enters a fault/error execution. The component of a typed language keeping track of types assigned to different program variables, and in general of expressions in the program and checking if an operation being executed is valid over a typed program variable, is termed as the *type system* of the typed language. A typed language can either be *explicitly* typed, if types are allowed as the part of the language syntax or *implicitly* typed otherwise. Most real-world languages are explicitly typed However, languages like ML or Haskell allow omitting some type annotations which can be inferred (explained later) by their type systems.

2.3.3 Execution Errors and Safety

Execution error of a program can either cause the program to terminate (abnormally) immediately or may go unnoticed and later cause arbitrary behavior. The former kind of errors are termed as *trapped errors*, while the latter are called *untrapped errors*. An example of an untrapped error is improperly accessing a legal address, like accessing a data outside the range of a data structure or jumping to a wrong memory address. Both these examples for few of the most exploitable vulnerabilities opening numerous attack windows. An example of a trapped error is calling an undefined procedure, division by zero, etc. A program or a program fragment is *safe* if it guarantees absence of untrapped errors. A language where every program fragment is guaranteed to be safe is called a *safe language*. Types and type systems are crucial tools for providing such language safety in the languages. In the absence of types, safety can be enforced via runtime checks.

2.3.4 Well-behaved Programs

For a given program, a set of trapped and untrapped errors form a set of *unsafe or forbidden errors*. A program is called *well-behaved*, if it does not allow any forbidden error to occur. A

well-behaved program or program fragment is *safe*. A language where all legal programs are well-behaved is called a *strongly-checked* language and when this strong checking is provided by a type system, the language is termed as strongly typed.

The well-behavior of a program fragment in a typed language can be enforced by performing static checks. These languages are called *statically checked* languages. The process of checking the safety or well-behavior is called *typechecking* and the algorithm which performs this type-checking is called the *typechecker* or the *typechecking algorithm*. This is achieved by checking that each program variable in the program is *well-typed* according to the set of rules for language expressions. These rules are termed as the *typechecking rules* of the type system. Some of the examples of statically checked languages are ML, Java, Pascal, Haskell, etc.

Besides static checking, typed or untyped languages can provide well-behavior guarantees including safety via detailed runtime checks in the programs. These runtime checks may exclude all forbidden errors in the program. This runtime checking process is called *dynamic checking* and the languages are called *dynamically checked languages*. LISP is a common example of dynamically checked language.

Most of the real world languages require dynamic checks along with the static checking, thus a language with static checking does not exclude dynamic checking.

2.3.5 Static and Dynamic Typing

Statically typed programming languages do type checking (the process of verifying and enforcing the constraints of types) at compile-time as opposed to run-time. Dynamically typed programming languages do type checking at run-time as opposed to Compile-time. Static type checking (static typing) is more common as it tries to approximate the runtime behavior of the program without incurring runtime cost. Static typing is present in many important languages like C, C++, Haskell, ML, Java, etc., while Scheme, Python, etc., belong to dynamically typed languages. Although less costly and efficient in catching errors early at compile time, static type systems are necessarily also conservative. They can categorically prove the absence of some bad program behaviors, but they cannot prove their presence, and hence they must also sometimes reject programs that actually behave well at runtime. For example, a program like-

if <complex test> *then* 5 *else* <type error>

will be rejected as ill-typed, even if it happens that the <complex test> will always evaluate to true, because a static analysis cannot determine that this is the case. The tension between conservativity and expressiveness is a fundamental fact of life in the design of type systems.

The desire to allow more programs to be typed by assigning more accurate types to their parts is the main force driving research in the field.

2.3.6 Typechecking Algorithm

An algorithm which checks that a program fragment is well-behaved is called a typechecker or a typechecking algorithm. More comprehensively, given a program variable or expression x, a *typing environment* Γ assigning types to each free variable in x, and a type τ, and a set of *typing rules* (explained later), a typechecking algorithm checks that τ is a correct type for x or can be assigned a type τ using the typing rules for the language.

There are various typechecking algorithms in the literature based on the feature and complexities of the underlying language, the property being checked and the typing rules. The typechecking algorithm used by us in the later chapters for typechecking p-typestate programs is based on typechecking algorithms for *dependently typed languages* [38].

Here we present a brief outline of the basic typechecker, which we extend to design our constraint-based dependent typechecking algorithm.

The basic typechecking algorithm takes a type annotated program or program fragment, language definition and a set of typing rules as input. It checks that for a set of expressions and statements called the *basic expressions*, the type of the value assigned to the expression is according to the annotated type and the typing rules. For *statements* without any return value, the typechecker checks that the statement is well-formed (according to the typing rules). It then uses these well-typed *basic expressions* to inductively check more complex expressions and statements. The algorithm has different cases, one for each expression or statement in the language definition. For instance, for a simple *assignment* statement of the form $x = e;$, the typechecker checks that the type of the value returned by the expression e is equivalent to (or subtype of, as explained later) the type of the variable x.

Chapter 5 presents a typechecker which extends this basic typechecking algorithm to typecheck complex p-typestate properties over a language with dependent types and objects. The algorithm rather than explicitly checking the correctness of assigned type for each expression, generates a set of *logical constraints* of the following form:

$$\forall x, y \in \mathbb{N}.(x \geq y \wedge x = 0 \wedge y = 0 \wedge ...)$$

The algorithm uses the typechecking rules (or what is called as typing rules) to generate these constraints, and later verify these constraints using an off-the-shelf SMT solver [43].

2.3.7 The Type Systems Language

A type system specifies typing rules for a language independently of any typecheking algorithm. This includes defining *types* for the *terms* of the language and rules for typing *expressions* of the language. These rules are themselves specified in a formal language having a specific semantics, here we explain this language of type systems.

Judgments A type judgment defines the type of a term or an expression in an environment Γ. A typical judgment is of the form:

$$\Gamma \vdash t : \tau$$

The above judgment is read as Γ entails that t has an associated type τ. A judgment of the above form specifies that a language term t has a type τ in a static typing environment Γ. A typing environment is a possibly-empty map (denoted by $\cdot$) of free variables of the language to types. *free variables* are the program variable whose type does not depend on the type of other variables as it is defined independently, for example, in the expression $\lambda x.x + y$, y is a free variable while x is a *bound variable*. Following examples show a set of judgment rules :

$$\cdot \vdash true : Bool \quad true \text{ is a constant of type } Bool$$

$$., x : Nat \vdash x + 1 : Nat \quad if \text{ x has a type } Nat \text{ then x+1 is of type } Nat$$

Another common form of typing judgment is the *well-formedness* judgment. A well-formedness property checks that a structure (like an environment, Class, etc.) is properly constructed. The well-formedness is represented by a special symbol $\star$. For example a possible representation of the well-formedness judgment for the typing environment Γ can be as follows:

$$\Gamma \vdash \star$$

Type Rules Type rules for a language check the correctness of certain typing judgment based on the correctness of other judgments. This is mainly achieved by inductive checking with some base cases which are intrinsically known to be true. A typing type rule is as follows:

$$\text{T-Rule Name}\frac{\Gamma_1 \vdash t_i : \tau_i \qquad \Gamma_2 \vdash t_j : \tau_j ... \Gamma_n \vdash t_n : \tau_n}{\Gamma \vdash t : \tau}$$

Each such type rule is written as a set of *premises*, judgments of the form $\Gamma_i \vdash t_i : \tau_i$, written above the horizontal line and a single *conclusion* judgment written below the line. Such a rule states that, if all the judgments in the premise are true, then the conclusion is true. If

a judgment's truth does not depend on any other judgments (an axiom), the premise is left empty.

We use an example of *simply typed lambda calculus*, the simplest typed language to demonstrate the structure and usage of the typesystem and typing rules. to define appropriate type rules we should begin with formally defining the underlying language called the *simply typed lambda calculus*. The language *Syntax* is defined in Table 2.1 and the *Evaluation* rules are shown in Figure 2.6 :

Syntax			
t	:=		terms
		x	variable
		$\lambda x : T.t$	abstraction
		t t	application
v	:=		values
		$\lambda x : T.t$	abstraction
T	:=		types
		T $\rightarrow$ T	function types

Table 2.1: Simply Typed Lambda Calculus Syntax

$$\text{E-APP1}\frac{t1 \rightarrow t1'}{t1\ t2 \rightarrow t1'\ t2}$$

$$\text{E-APP2}\frac{t2 \rightarrow t2'}{v1\ t2 \rightarrow v1\ t2'}$$

$$\text{E-APPABS}\frac{}{(\lambda x : T_{11}.t2)v2 \rightarrow [x \mapsto v2]t12}$$

Figure 2.6: Evaluation $t \rightarrow t'$

A term t in the language is either a variable (x), or an abstraction, shown as a *lambda* term, or an application of a term to another. Abstraction is called a *lambda term* and it represents a *function* which is the most basic building block of *lambda calculus*. A lambda term has a bound variable x and an annotated type T, and another term t representing the body of the

abstraction. The application contains two terms, one applied to another. The *Evaluation* rules define small step semantics for the language, showing how a term t evolves to t'. It is worth noticing that rule (E-APP2) applies only if the first term of an application is an abstraction. Intuitively, it says, for a given function term $v1$, if its parameter reduces from $t2$ to $t2'$, then the application is performed on the reduced term. Rule (E-APPABS) presents a reduction rule for application (called the β reduction in literature), the rule says that the parameter x of the LHS abstraction is replaced by the actual value in the body term of the abstraction. This is an example of call-by-value evaluation semantics.

Next, we present typing rules to enforce well-typedness of programs, there is a type rule for each term in the language. As mentioned earlier, a typechecking algorithm checks that a program fragment satisfies these typing rules.

$$\text{T-VAR}\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{T-ABS}\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

$$\text{T-APP}\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

These type rules define well-behaved expressions, the (T-VAR) rule say that the type of a free variable x is same as its type in the typing environment. The (T-ABS) rule defines the good behavior for abstractions. It specified that, if in an extended typing environment, extending the typing environment with a mapping for the bounded variable x to T_1, the type of the body is T_2, then the type of the abstraction term is $T_1 \to T_2$. The (T-APP) rule defines a well-behaved application expression. It specifies that for a given *abstraction value* t_1 of the type $T_{11} \to T_{12}$ and a given parameter term t_2 of the type T_{11}, the return type of the application must satisfy the type of the abstraction.

This short introduction to syntax, evaluation rules and the typing rules will help elaborate the contribution and ideas discussed for p-typestate type systems in Chapter 5.

2.3.8 Type Equivalence and Subtyping

Typechecking is a process of verifying that each term in the program is well-typed based on the typing rules of the type-system. Typecheking can be done either at compile time or at runtime, thus defining *static* and *dynamic* typechecking respectively. Checking the correctness of associated types further requires checking if two types are equal or equivalent. Generally,

there are two kinds of *type equivalences*, *viz.*, *Nominal* and *Structural*, based on two kinds of type systems of the same name. Type systems like Javas, in which names are significant and equivalence and subtyping are explicitly declared, are called nominal. Others in which names are inessential and subtyping is defined directly on the structures of types are called structural. The details of nominal and structural type systems and their equivalences and subtyping will require extensive discussion which is beyond the scope of this work. The details can be found in a standard text on type systems, such as [105].

Two types τ_1 and τ_2 are equivalent if one can be replaced by the other in the type system. The following typing rule defines this definition formally:

$$\text{T-Equivalence}\frac{\Gamma \vdash \tau_1 \simeq \tau_2 \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash t_1 : \tau_2}$$

One important expected property of a good type system is that it should be *decidably verifiable* , i.e., the *typechecking* algorithm should always terminate. There is always a trade-off between the decidability of the typechecking problem and the expressiveness of the type system. For instance, a rich type system that allows rich predicate logical expressions as a part of the type annotations has high expressiveness but may lack decidability while simpler type systems have a decidable and efficient type checking but lack expressiveness to check the correctness of rich program properties.

The equivalence of types is a relatively stronger restriction for types to be replaceable. Thus many type systems allow *subtyping*, a much weaker relation than equivalence to be sufficient for replacement of types. A subtyping relationship, represented typically as $\tau_1 <: \tau_2$ (i.e. τ_1 is a subtype of τ_2) allows τ_1 to replace τ_2 in typing rules of the type system. Thus subtyping relation captures the fact that a subtype is more refined or contains more information than the supertype and hence can safely be used in a context where a term of its supertype is expected. This is termed as "the principle of safe substitution". Subtyping relation can be added to the type systems by adding three main extra typing rules. One formally defines the principle of safe substitution while other two capture the general *reflexive* and *transitive* properties of the subtyping relation.

$$\text{T-subtyping}\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_2 : \tau_1}$$

$$\text{T-Ref}\frac{}{\vdash \tau <: \tau}$$

$$\text{T-Trans}\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma\tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}$$

2.3.9 Type Inference

Annotating each term of a program with its type is a hard task even for experienced programmers and this burden could be placated by automatically deducting the *most general* possible type of a term in the program. This process of automatic deduction of the most general type of a term in a program is called *type inference*. Type inference is a much harder problem than typechecking. The type inference problem for a given term is dependent on the type system in question. Thus a simple type system cannot express many useful properties but has a much easier type inference problem to solve as compared to a richer type system with expressive power. For instance, the type inference problem for an explicitly typed language like Pascal [114] is easy to solve as compared to an implicitly typed language like ML [91]. The type inference problem is especially hard with richer type systems like one with *polymorphic types* [29] and other type theories like *dependent types*.

2.3.10 Dependent Types

In type theory, a dependent type is a type which depends or is a function of value/term of some other type. Thus, it is dependent on the terms of other type and hence the name. A classic example of dependent types is $('a\ Vector\ n)$, *Vectors* of size n and component type a. Such a *Vector* can be defined using a *type family* of *Vectors*, which are basically one-dimensional sized arrays. A *type family* is defined as :

$$Vector : Nat \rightarrow \text{Type}$$

The above declaration asserts that $Vector$ is a *type constructor* which maps a natural number $k : Nat$ to a type. The idea is that the type $Vectork$ contains vectors of length k of elements of some fixed type, say *data*.

To use these types which are functions of terms, we need to introduce them (instantiate an element of the *type family*). For example, a way to initialize the *Vector*, is to define a function, init which takes a type a of data and an $n : Nat$ and returns a vector of components a and size n. In dependent type theory, the typing of this function will generally be written as,

$$\mathsf{init} : \Pi n : Nat.\ a \rightarrow' a\ Vector\ n$$

The type of init introduces the *dependent product type* ("Pi type"), represented as $\Pi x : S.\mathrm{T}$. This is a dependent function type, a generalization of the function type $(T \rightarrow T)$ of the simply type lambda calculus [105]. It is a function that maps element $s : S$ to a Type generated by

$[x \mapsto s]T$. Compared to the simple *function types*, the result type of a function with a Π-type can vary according to the argument supplied.

Expressiveness and Usage of Dependent Types The expressive power of dependent types comes from its capabilities to capture dynamic properties of the data in its type, like the length of the *Vector* data structure in the above example. Other example are *matrices* of size $m \times n$, *trees* of height m or *height balanced trees* of certain height. This makes it possible to check some of the dynamic properties of the program statically. For example, such dependently typed Lists and Arrays with their size information a part of their definition/type may eliminate the need for runtime array-bound checks [127]. This can be achieved by defining a function to access the element on following lines:

$$getElement : \Pi(size : Nat).Pi(i : int \mid \{0 \leq i < size\}).Int(i) \rightarrow List(size) \rightarrow' a$$

The function's signature takes as parameter an integer index i, which is restricted to be strictly less than the size *size* of the List. The function is not defined for indexes larger than the size of the List, and thus it is guaranteed to avoid accesses beyond the size of the List. In general, the *parameters* to a dependent type can be any term of the language (including a whole program). This gives dependent types their extreme expressive power, but as the complexity of the parameters increase, reasoning and typechecking the properties captured by them (like the property of the size or the index range mentioned above) becomes complex. In general, the typechecking problem for dependent types is undecidable [44]. Thus, in theory, dependent types can statically verify any property (structural or dynamic) of interest, but this expressiveness is bounded by the undecidability results of dependent typechecking (discussed later) and thus most of the practical languages and systems using dependent types to model and verify program properties restrict the type system in various ways to achieve an expressive yet decidable type system.

While there are numerous works [127, 26, 98] exploring the practical utility of dependent types in the *functional languages* domain, the applicability in imperative languages domain is less explored [126] and negligible in the domain of Object-oriented languages [99]. This may be attributed to the hardness of reasoning about *side-effects* and *states* and their effect on dependent types. Breaking this pattern, we present a dependently-typed language including p-typestates which is an imperative, typestate-oriented language and object-oriented languages.

Besides Π types, the dependent type theory also presents a generalization of *pairs* or *tuples* called as the Σ types. The sigma types, represented as $\Sigma x : T_1.T_2$ generalize ordinary products (A × B), just as dependent function type Π generalizes ordinary functions. A sigma type

$\Sigma x : T_1.T_2$, represents a type family $(x : T_1, y : T_2)$, such that type of y depends on x. The degenerate case when x does not appear free in T_2 amounts to the ordinary product type written as $T_1 \times T_2$. In this thesis, we use only Π types to model p-typestates and thus we do not elaborate further on *Sigma* types. The interested reader may refer a standard work on dependent types like [106].

Dependent function types and Sigma types can capture various dynamic or behavioral properties of data and program using type parameters which can be any valid term in the language. For example, dependent types can be a good candidate to capture the state associated with an *object* in an object-oriented language as an interface. This idea is similar to typestates and thus makes dependent types particularly useful to express typestate properties and checking them statically. For example, consider the typestate property associated with an *Iterator* over a Java *Collection*. The *next* method is valid only if the Collection has more elements, and this is generally checked dynamically (for example in Java Collections) using a method *hasNext*. This property can be statically enforced by enumerating the states of the *Iterator* and then defining methods, with dependently typed parameters, parameterized over the states. Listing 2.5 shows an Interface for such an *Iterator*. The definition guarantees that we can never call next and remove in an illegal state. The listing declares a set of states IterState = { S, T, U, V } (names of the states are not significant) and creates an Iterator as a dependent function type, indexed over this set. This allows a programmer to define functions like hasNext, next, remove over this Iterator. For, example, function hasNext is defined only for an Iterator is a state S or U and not for other two states. Thus, using dependent types a programmer can define when an operation is valid/invalid over a given data object, or in other words encode *typestates*. Such an approach of encoding typestates using dependent types has been discussed in some of the works in the functional languages domain. For example, *Eff* [25] discusses how *effects* can be modeled and reasoned about using dependent types in *Idris* [26].

Listing 2.5: Iterator interface using dependent types

```
1 IterState = {S, T, U, V };
2 Iterator : IterState -> Type
3 hasNext : Πst {S, U }. Iterator(st) -> ΣnewSt {S, T, V }.Iterator(newSt)
4 next : Πst {T, V }. Iterator(st) -> ΠnewSt {S, U }.Iterator(newSt)
5 remove : Πst {U, V }. Iterator(st) -> ΣnewSt {S, U }.Iterator(newSt)
```

2.3.11 Curry-Howard Correspondence

Another way to introduce dependent types is through *Curry-Howard correspondence*, also known as *proposition-as-types*. The correspondence associates simple types to propositions in a *constructive logic theory* [93]. A term of a type is analogous to a proof of a proposition.

The underlying idea is the relationship between types, terms, functions, etc. to propositions, proofs and implications respectively. This correspondence could be further extended to first-order predicate logic leading to dependent types. A more detailed discussion of this topic is beyond the scope of this thesis and the reader is referred to [122]

2.4 Presburger Arithmetic Logic

As discussed above, dependent types' expressiveness may be utilized to encode typestate properties. The same technique can further be used for modeling and verifying richer typestate properties. The *p-typestate* type system discussed in Chapter 5 employs this concept. Unfortunately, typechecking cannot be automated for the most expressive general dependent types as mentioned before, thus we restrict the language for dependent types' *parameters* to a decidable logical family of Presburger arithmetic formulas. In this section, we present a brief introduction to this family which will aid in the elaboration of the ideas discussed in Chapter 5.

A Presburger Logic is a *First-Order* [90] Logic of $(\mathbb{N}, <, +)$ interpreted over $\mathbb{N} = \{0, 1, 2, ...\}$. The Logic was first studied extensively by Mojzesz Presburger in 1929 who gave a sound and complete axiomatization and a procedure for the validity of the logic. Presburger logic allows defining formulas over the set of Natural numbers with addition and without multiplication. For instance, we can have formulas like:

$$x + 3y + z < z + 1,\ \forall x \forall y((x < y) \Rightarrow \exists z(x < y < z)$$

The logic is not as expressive as Peano arithmetic [100], but this simplicity makes *validity* and *satisfiability* problems for the class of Presburger arithmetic decidable [108], i.e. there exists an algorithm that can decide if any given statement in Presburger arithmetic is true. No such algorithm exists for the richer Peano arithmetic.

A Presburger arithmetic logical formula is formally defined as follows:

(formula)	ϕ	:=	$\phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists var.\phi$
			$\mid \forall var.\phi \mid$ t $\sharp$ t
(term)	t	:=	n $\mid$ t + t $\mid$ - t $\mid$ n * t $\mid$ var
(relop)	$\sharp$	:=	$< \mid > \mid \leq \mid \geq \mid =$
(var)	var	:=	x $\mid$ y $\mid$ z ...
(numeral)	n	:=	0 $\mid$ 1 $\mid$ 2 ...

Table 2.2: Presburger Formula

2.4.1 Decision Properties and Procedures

A decision procedure for Presburger arithmetic is an algorithm to decide the validity of a Presburger formula. If an integer formula is free from quantifiers, its truth value can be easily calculated. There are various quantifier elimination procedures for Presburger arithmetic like Fourier-Motzkin, Omega test, Cooper algorithm etc. In our work, we use the Z3 theorem prover [43], which has efficient implementations of these basic and other advanced algorithms for deciding the validity of Presburger arithmetic formulas. The theory of Presburger arithmetic is available at [108].

2.4.2 Typestates

Type systems define the set of valid or restricted operations on data. While this helps to prevent many programming errors or proving the absence of a certain kind of programming errors, there are many programming errors that types cannot capture. One subset of such programming errors are those which are caused due to an incorrect state (a typical *IllegalStateException* in Java) or an incorrect ordering of method calls (or function calls) over some data, for instance calling a *read* operation on a File object which is not *open*. In the absence of a static type checking system, these errors are mostly checked at runtime. For instance Java compiler checks for an *IllegalStateException* and *Array bounds checks* at runtime. Typestates [115], are designed to capture these kinds of programming errors or defects by adding a *state* component to the normal types. Thus, while types define what operations are allowed or restricted on data for its lifetime, typestates define what subset of these operations are allowed or restricted in a given state. For instance, a typestate including the state of a File object (*open, close*) may let us define a temporal ordering of properties requiring the *read* operation being called only in an *open* state.

Another example property that requires typestate modeling, and is very common in object-oriented programs is related to *Iterators* over *Collections*, like Iterator over a List object in Java. A List iterator has various operations associated with it, like a boolean returning method *hasNext()* and another method *next()*, returning the next element of the list. Now an Iterator class in Java places a restriction/protocol on its usage- the *next()* method may only be called when there are more elements in the container on which the Iterator is iterating, else a *NoSuchElementException* is raised. This is an example of erroneous usage of an API by the programmer. Such usage protocols need to capture the states of the objects or the environment and could be readily modeled and verified using Typestates.

The problem is acerbated in the presence of *aliasing*. Precise reasoning and tracking of state

changes in such a system must be aware of all the aliases to a reference, in order to soundly track the current state. This is a challenging problem, as the state change occurring in one context via a reference may not be visible to other aliasing references in some other context. For instance, one alias might be depleting an iterator while another might still believe that the iterator has a value through a previous call to hasNext(). Such errors are really hard to debug, as failure is separated from the point of detection by the runtime system in both space and time, and the stack trace emitted for the failure provides no information on the other calls that have been made recently on the object, or why the local assumptions about the state of the object are incorrect. There are some useful works [53] which integrate a static *alias* analysis with static typestate analysis. The idea is to have a sound (maybe imprecise) alias analysis feed the aliasing information to the typestate analysis. This solves the aliasing problem discussed above. There are other approaches for managing aliases and capturing them statically, we discuss one such approach based on *permission* system in Chapter 5. Thus typestate is an important programming language concept to track and verify regular, private state and state mutation related properties of programs which could not be captured using constant non-mutating type systems.

2.4.3 Counter Automata/Machines

A k-counter automaton is a finite automaton augmented with k integer counters associated with the machine. The machine has a two-way read-only input head. In each transition, the machine can independently increment or decrement the counters and compare them against 0 and can move the input head left or right.

Definition 2.1 *Multiple Counter Automaton A multiple counter automaton is a tuple* $\langle Q, C, \delta \subseteq Q \times G(C, C') \times Q \rangle$ *where*

- *Q is a finite set of states with a distinct initial state.*
- *C is a finite set of counter variables (These can be integers or natural numbers). C' is the primed version of C representing the changed value of C.*
- *G (C, C') is the set of guards built on alphabet C, C'. A member of G (C, C') is a Presburger formula as defined earlier.*

Figure 2.7 shows an example of a simple counter system with a single control state q_0 and two transitions t_0 and t_1. Each transition has associated guards g_0 and g_1 respectively.

A *configuration* s of a multiple counter system is described not only by the control state of the system but also by the values of these finitely many counters associated with each

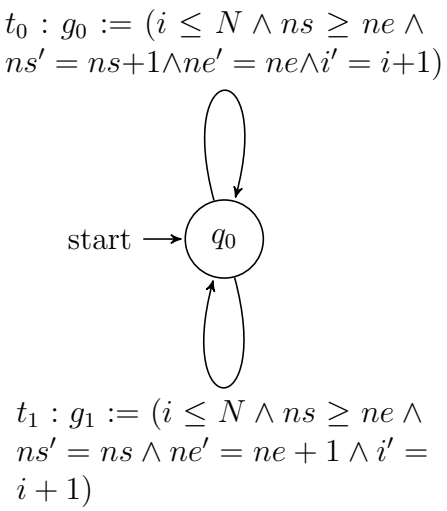


Figure 2.7: A multiple counter automaton with single state and two guarded transitions

state. For example, a possible configuration for the counter automaton of Figure 2.7 can be $(q_0, \{i = 0, ns = 2, ne = 1\})$. The transition relation δ, makes a transition based on the current configuration and the guards to a new configuration.

The semantics of a counter system is defined as follows. A transition $t \in \delta$ represented as $s \rightarrow s'$ over the states of the system is allowed if the state s satisfies the guards gt associated with the transition. The state s' (s) is termed as the immediate successor (predecessor) of s (s'). A counter system can be *deterministic* or *non-deterministic*, either due to a single transition or a set of transitions from a state.

What makes counter system/automaton so interesting is the expressive power of this machine. A *2-counter machine* (A counter system with two counters) can simulate a stack and since a two stack machine can simulate an arbitrary Turing machine, it follows that a 4-counter machine can simulate a Turing machine. This expressiveness comes at the cost of decidability. The set of reachable states for a multiple counter automaton is possibly infinite and hence cannot be enumerated. Thus finding the set of all the reachable states for such systems is undecidable in general, thus making normal approaches to *model checking* moot. *Safety analysis* of these systems require symbolic model checking.

2.5 Verification of Infinite State Systems

The p-typestates work discussed in Chapter 5 presents a novel loop invariant calculation for Presburger-definable loops. The approach uses *acceleration* techniques from the *symbolic model checking* domain of infinite state systems. In this section, we present the required background

knowledge about model-checking, symbolic model-checking, verification of infinite state systems, etc. These ideas will aid in better comprehension of the underlying principles of the loop invariant calculation approach discussed in Chapter 5.

2.5.1 Model Checking

Formal verification methods are used to prove the correctness of software systems. This aids in reducing the high cost of correcting errors in these systems at the same time provide guarantees crucial for safety-critical systems. There are many formal methods of verifying systems [35, 105, 45] and each of these methods have three basic elements, a mathematical model of the system to be verified, a formal language for specifying and formulating the correct and incorrect behavior of the system and a procedure or an algorithm to verify the correctness of the system.

Model checking is one such formal method of verification, introduced by Clarke and Emmerson [35]. Model checking determines the truth value of a formula representing the correct or incorrect behavior in a specific finite model of the system. This requires an exhaustive search of the explicit state space of the system. The number of explicit concrete states in the model of many systems (like concurrent systems) can grow exponentially. This is termed as the *state space explosion problem*, which makes explicit state exploration techniques ill-suited for infinite-state systems. Since, a large fraction of real-world systems and properties of interest are beyond the finite state model, verifying safety and other correctness properties of infinite-state systems is an active area of research [27]. These research works allow to verify properties of models like *pushdown systems* [49, 21], *counter systems* [37, 54], etc. One such solution in the literature to handle the state space explosion problem is *symbolic model checking* [15].

2.5.2 Verification of Infinite State Systems

The explicit state space exploration approach of model checking is not applicable for *infinite state systems*. Thus verification approaches for such systems must be based on *symbolic execution* since it is possible to manipulate some finite symbolic representation of the infinite state space. This approach is called, *symbolic model checking*. There are various tools for checking the reachability properties of infinite-state counter systems based on symbolic model checking. [54, 11]. Symbolic model checking for infinite systems although tractable is hindered by the problem of *convergence*. The next subsection describes the problem and presents a technique to handle the problem effectively.

2.5.3 Acceleration

The problem of calculating the set of reachable states (REACH) for an infinite state system is undecidable in general. Thus, model checking infinite state systems uses a "symbolic" approach which involves abstracting a symbolic model of the model checking problem and manipulating it to calculate fixpoints for forward and backward reachability sets. A naive fixpoint calculation for these infinite systems may diverge in general and thus has a low probability of termination. Acceleration [15] is a popular technique which makes the convergence of the fixpoint calculation for such systems more frequent. The technique is analogous to abstract widening operation from the abstract interpretation domain.

Definition 2.2 (Acceleration over a path π) Given a transition system $\mathbb{T} = \langle Q, \Sigma, \Psi, \delta \rangle$ and a sequence of action $\pi \in \Sigma^*$. *Acceleration* of π over $\mathbb{T}$ is called π -acceleration and is defined as a relation $Acc_\pi \subseteq (Q \times Q)$ such that (s, s') $\in Acc_\pi$ iff $\exists k \in \mathbb{N}$. such that s $\xrightarrow{\pi^k}$ s', where s $\xrightarrow{\pi^k}$ s' represents a path (s $\xrightarrow{e_1}$ s1 $\xrightarrow{e_2}$ s2... sk-1 $\xrightarrow{e_k}$ s') of k consecutive transitions in the system in δ. We say that s' $\in post_\mathbb{T}(\pi^*, s)$ or simply $post^*(s)$, where $post_\mathbb{T}(\pi^*, s)$ represents the set of post reachable states by the acceleration of π over $\mathbb{T}$, starting from the initial state s. The definition could be extended to a set of starting states S, by calculating $post^*(s), \forall s \in S$. The acceleration relation Acc_π is called π acceleration or just acceleration when the context is obvious.

According to the property of the paths being accelerated, an acceleration defined above can be refined to either *loop acceleration*, *flat acceleration* or *global acceleration*. The details of the acceleration techniques can be found in [54]. A model or a system supporting a global acceleration implies that it supports flat and loop acceleration. Similarly, supporting flat acceleration implies support for loop acceleration. Accordingly, the complexity of the procedure for their computation grows contra-variant. Please refer Chapter 5 for a more detailed description of acceleration techniques and its application to loop-invariant calculation.

We model the loop invariant calculation for a program enforcing a p-typestate property as REACH finding problem over the counter system induced by the looping construct in the program. This reduction allows us to use known acceleration based reachable states computation approaches and tools. The reduction is straightforward, Presburger formulas over integer variables of the counter system for the input loop forms the symbolic domain. The acceleration of the loop calculates a Presburger definable formula representing the REACH set for the input loop with the given initial set of states. We use this formula as a loop invariant in p-typestate type checking to generate a modular proof of correctness of the program.

2.6 Chapter Summary

In this chapter, we presented a brief introduction to some of the important preliminary and background required to easily understand the technical contributions and ideas discussed in the rest of the thesis. We began with a brief background of Android applications, static analysis in general, and particularly in the context of an Android application. This was followed by a brief introduction to a background on *types*, *type systems* and richer type theories like *dependent types*. We also discussed mathematical logic and logical families like Presburegr arithmetic. This was followed by a small discussion about verification of infinite-state systems. These background topics will help to understand the ideas discussed in Chapter 4 and Chapter 5.

Chapter 3

Related Work

We propose programming language based, static approaches to verify rich typestate-like safety properties over complex programs in this thesis. This includes two major components, first, an asynchrony aware static analysis for event-driven programs like Android applications and second, a generalized notion of typestate, called parameterized typestates (p-typestates) to specify and verify typestate properties beyond the regular language domain. We mentioned these contributions in Chapter 1 by depicting them along two axes of "richness of typestate property (ϕ)" and the "complexity of the programs being verified (P)". In this chapter, we discuss the most closely related works along both these axes and a few others which are related to the general static analysis, type systems, typestate analysis and analysis of Android applications. This is by no means an exhaustive list of related works but gives sufficient understanding of the state-of-the-art in research in related fields.

3.1 Modeling and Static Analysis of Android Applications

3.1.1 Operational Semantics for Android Activities

Payet et al. [103] present operational semantics for Android *Activities* along with the semantics for the *Dalvik instructions*. The work defines a core simplified Dalvik instruction set and presents the semantics as transformations over states of the Dalvik Virtual Machine. Each state of the machine is depicted as a tuple $\langle r \mid \pi \mid \mu \rangle$ of *registers*, *stack* of pending activity calls and *heap* respectively. A heap representation maps locations into *objects*. An object is a function that maps its fields into values and that embeds a class tag k. Such an object is called to be belonging to class k.

Apart from the basic instructions set, they further define a set of *macro-instructions* which tries to capture the semantics of Android Framework API methods like *startActivity, findViewById, setResult*, etc. The semantics of these macro-instructions are defined in terms of state transitions. An activity has an associated *state*, which basically captures which lifecycle callback of the activity is being executed. An activity state is an element of the set containing all possible lifecycle callback methods of the *Activity* class.

$$\{constructor, onCreate, onStart, onRestart, onResume..., onDestroy\}$$

Lifecycles associated with an activity is represented as a set of binary relations over these activity states. An activity frame is a tuple $\langle l \mid s \mid \pi \mid \alpha \rangle$ used to manage an activity a in an activity stack. l is the location of a on the heap, s is the current state of a, π is an activity set defining activities that are waiting to be launched from a and α is a method stack used for managing the execution of a callback method corresponding to s.

This work presents the first formal semantics for selected Android control flow features. This can be utilized by static analyses to precisely capture the lifecycle semantics of Android applications. However, the semantics presented in the work have certain limitations. The semantics has no way of capturing the asynchronous nature of ICC in Android, and the interaction with the Android system is also imprecisely modeled. For example, although representing lifecycles as binary relations and then defining semantics for lifecycle, captures how the control might flow in a component, the operational semantics are not well designed to capture precise control flow semantics caused by the interaction between lifecycle callbacks and the inter-component communications (ICC). Correctly modeling such interactions is crucial for a precise static analysis of Android apps. Further, the semantics for other application components cannot be defined easily using the given semantics. For instance, the set of *macro-instructions*, and their semantics, presented in the work, are not complete and newer instruction sets are needed separately for ICC and lifecycle API of other component types. Extensive usage of API for control flow in Android applications makes it a challenging task to completely capture Android application control flow using this approach. This makes the formal semantics too verbose which might be as difficult to reason about as the original application. A simple approach can be to have a common small, abstract set of ICC and life-cycle semantic rules for each component types and modeling the control flows using an intermediate control flow graph representation. This is the approach taken by our asynchrony-aware static analysis work discussed in Chapter 4.

3.1.2 Formal Modeling and Reasoning about Android Security Framework

Armando et al. [12] presents a formal model for the *Android Security Framework* or the *Android Security Architecture*. The work models the Android architecture, i.e., the Android *stack* (refer Chapter 2), Android applications, and interactions between applications and the Android Framework. This provides a language needed to describe security-relevant aspects of the Android Security Framework. The work provides a high-level abstraction for an Android application, its *components, resources, manifest, permissions, Intents*, granting and revoking dynamic permissions, etc.. It provides operational semantics presenting the computation of a Component in a given context. These constructs and associated semantics are suitable for modeling creation of resources, components, checking capabilities and other properties related to *access control* in Android applications. Unfortunately, the model cannot capture control flow semantics in Android applications, such as asynchronous call semantics, component lifecycles and other callbacks, and possible interaction between asynchronous calls, ICCs and lifecycles. The Android application and control flow semantics provided by us in Chapter 4 aptly capture these features in Android applications and aids in *sound* static analysis of these applications.

Fragkaki et al. [57] present a formal model of the Android permission system and a formal framework for analyzing and verifying Android style (permission based) security properties related to integrity and privacy of application and user's data. The work further presents an implementation of a property enforcement system called SORBET which provides the required language to specify flexible security policies and a system to enforce these policies. This work extensively increases the expressiveness of android access control based permission system by allowing to enforce richer *coarse grained security policies* related to *privilege escalation* [42] and information flow [13, 76, 46]. Our formal model and semantics of Android application is used to define an intermediate representation for Android applications which soundly captures all possible asynchronous control flows in Android applications.

3.1.3 Other Formal Studies of Android Applications

There is a plethora of works on formal modeling and studying/analyzing Android framework and applications. Smith et al. present a formal model of a subset of Android applications in the ACL2 theorem prover [94]. The work then builds and proves useful properties over these modeled applications. This work is more along the lines of another important formal work [67], which provides a formal model of Android applications and uses symbolic execution over the model to prove useful application properties. These works differ significantly from

our modeling of Android. Our Android application model explicitly captures the asynchronous control flow Android due to asynchronous calls, call-backs, lifecycles, and other framework induced asynchrony, while these works either try capturing the high level structure of Android applications, and its security framework, or are interested in modeling the Dalvik instruction level semantics to utilize it for symbolic execution. Further, none of these works captures the asynchrony semantics in Android, which is important to soundly and precisely track many useful properties over Android applications as exhibited by us empirically in Chapter 4. Chen et al. [32], try to capture the interaction between an Android application events and permission via an intermediate representation called *Permission Event Graphs*. This is an abstraction of the events and permissions in applications, allowing them to define and verify simple temporal properties over events and permissions. This work is orthogonal to our model of Android, and it will be interesting to check such properties using the typestate analysis developed by us over AICCFG. This will require adding abstractions for permissions in our model of Android control flow. Besides these, there are several other works on Android application modeling [58], but none of these works models the asynchronous control flow semantics in Android applications which are the major focus of our work.

3.1.4 Featherweight Java

Igarashi et al. [65] presents Featherweight Java (FWJ), a minimal core calculus for Java's type system. The work aims for the compactness of the calculus against the *completeness*, and captures the core of Java language and its type system, using five language expressions: object creation, method invocation, field access, casting, and variables. One key omission from FWJ is the *assignment*, each field and variable in FWJ is basically final and there is no *side-effect*. This reduces FWJ to the "functional" fragment of Java. FWJ presents the core concrete syntax, a simple type system and a static semantics (dropping side-effects allows one to define evaluation semantics completely in the syntax of FWJ) for this functional fragment of Java. The major aim of FWJ is two-fold. Firstly, it aims to provide a core language, over which bigger languages can be designed with richer features. For example, FWJ work also presents an extension of FWJ with *Generics* [101] called the Featherweight Generics Java (FGJ). Secondly, it aims at providing a minimal and concise type system which can help reasoning over these programs, and which has a concise proof of *soundness*. Further, since the proof of soundness for pure FWJ is very simple, a rigorous soundness proof for even a significant extension may remain manageable. The work gives type soundness proofs for FWJ and FGJ. FWJ has aided in the design of the typestate-oriented programming language [9] (FTS). We extend this typestate-oriented programming with our parameterized typestates and related machinery of dependent

types over Presburger-defined formulas.

3.2 Static Analysis of Android Applications

3.2.1 Intra-component Information Flow Analysis

FlowDroid [13] is an intra-component *information flow analysis* for Android applications which is *context, flow, field and object sensitive* in traditional static analysis sense. The analysis assumes that all information passed across ICC is tainted, and over-approximates asynchronous calls across components by assuming any possible ordering of component interactions. FlowDroid models the lifecycle associated with each component but misses some of the possible control flows occurring due to the interactions between the lifecycles of components. FlowDroid presents an algorithm for precise taint analysis coupled with an on-demand alias analysis as an extension of the IFDS interprocedural analysis algorithm [110]. The algorithm has two tightly coupled solvers, a forward solver for *taint analysis* and a backward solver, which is triggered on-demand for *a precise alias analysis*. The paper also presents a bunch of benchmark applications called *DroidBench* which contains test cases for interesting static taint and other related analysis problems. In summary, the FlowDroid tool takes an application package, parses the manifest, the *dex* files and the layouts to generate a *main* method making synchronous calls to each component to generate a call-graph, and uses its forward and backward running algorithms for taint analysis. The major limitation of this analysis is its inability to handle inter-component flows as most of the real world applications have ICC across components. The over-approximate solution is highly imprecise for a property like typestates which requires a more precise flow-sensitive analysis as compared to a *taint analysis*. Moreover, the control flow graph generated by the approach for an application misses certain possible applications control flows, making it unsound (refer Chapter 4 for the definition of Soundness) as well.

3.2.2 Inter-component Information Flow Analysis

IccTA [76], extends the FlowDroid tool with inter-component communication analysis to find *privacy leaks*. It tries to improvise over intra-component analysis of FlowDroid. IccTA calculates or extracts ICC calls and their targets (calling it *ICC links*) in the application and stores these in a database. Based on these ICC links, IccTA instruments the original input application with explicit calls to the target component at the ICC location. Let us consider an example code fragment from the paper that handles the *startActivity* ICC method.

Figure 3.1 shows the original application code with the ICC call startActivity in line 13. Figures 3.2, 3.3 shows the instrumented version of the application with a call to the instru-

```
Intent myIntent = new Intent (action, data);
startActivity(myIntent);
 //TelephonyManager telMnger; (default)
 //SmsManager sms; (default)
class Activity1 extends Activity {
  void onCreate(Bundle state) {
    Button to2 = (Button) findViewById(to2a);
    to2.setOnClickListener(new OnClickListener(){
  void onClick(View v) {
    String id = telMnger.getDeviceId();
    Intent i = new Intent(Activity1.this,Activity2.class);
    i.putExtra("sensitive", id);
    Activity1.this.startActivity(i);
  }
 });}}
class Activity2 extends Activity {
  void onStart() {
    Intent i = getIntent();
    String s = i.getStringExtra("sensitive");
    sms.sendTextMessage(number,null,s,null,null);
 }
}
```

Figure 3.1: An example application with ICC

```
// modifications of Activity1
Activity1.this.startActivity(i);
IpcSC.redirect0(i);


// creation of a helper class
  class IpcSC {
    static void redirect0(Intent i) {
    Activity2 a2 = new Activity2(i);
    a2.dummyMain();
  }
}
```

Figure 3.2: Instrumentation by IccTA for ICC

```
// modifications in Activity2
  public Activity2(Intent i) {
      this.intent_for_ipc = i;
  }
  public Intent getIntent() {
    return this.intent_for_ipc;
  }
  public void dummyMain() {
   // lifecycle and callbacks
   // are called here
}
```

Figure 3.3: Instrumentation by IccTA for ICC

mentation method IpcSc.redirect0(i), which creates an explicit instance of the target component Activity2 and calls the *dummymain* (entry point for the component) for the target. This gives a *whole-application* control flow graph for the application, which IccTA uses to run a normal IFDS taint analysis. The work is closely related to another similar work, AmanDroid [125] which also presents a similar inter-component static analysis for Android applications. These works efficiently capture many privacy leaking paths in applications but suffer from serious limitations. Inter-component communications, and lifecycle and other callbacks in Android applications have *asynchronous* calling semantics. This means that the caller does not block for the callee to complete, and proceeds with its execution according to the non-preemptive lifecycle callback execution semantics while the asynchronously called method is queued for *dispatch* by the scheduler (*ActivityManager Service in Android*) and is executed some later time. Although these works capture these ICC calls and returns, they incorrectly model them as *synchronous* calls. The asynchronous call and return semantics become relevant to the soundness and precision when a static analysis needs to track a property of some global data as the value of the global data flow fact might change between the call to the target component and the actual dispatch. Moreover, the effect of unsoundness is magnified due to the interactions between the control flows due to ICC and lifecycles. We present a sound asynchrony-aware static analysis approach for Android applications to mitigate these and other limitations of state-of-the-art static analysis approach for Android. The details of the approach are discussed in Chapter 4.

3.2.3 Other Static Analyses for Android Applications

There is a plethora of other static analysis works for Android which prominently aims to extend normal static analyses for Java applications to Android applications. [58, 77], while a few others try to analyze Android applications for specific properties like inter-application communications [33], *applications collusion* attacks [128], *privilege escalation* attacks [42], etc. There are several good surveys [77] about static analysis for Android applications, and interested readers should follow some of these. All these works treat Android applications as a variant of Java applications with specific challenges to analyze. Although some of these as discussed here aim at modeling the life cycles of the components, none of these correctly model the asynchronous control flow semantics of these applications which is inherent in ICC and event callbacks.

3.3 Static Analysis of Asynchronous Programs

Jhala et al. [68] present a formalism for the interprocedural analysis of asynchronous programs using the IFDS [110] approach for similar analysis over *synchronous* programs. The approach

is called Asynchronous IFDS or AIFDS. For programs with no asynchronous function calls, the *interprocedural data flow analysis* [110, 80] forms a general framework for the program analysis. Unfortunately, the IFDS approach is not applicable to the programs with asynchronous function calls and returns. For instance, consider Figure 3.4 adapted from the paper, which shows an asynchronous program Plb called from an event-driven load balancer. Such asynchronous calls and callbacks are extremely popular for event driven programs like Android and other mobile OS applications. Execution begins in the procedure main which makes an asynchronous call to a procedure textsfreqs (line 4), that adds requests to the global request list r, and makes another asynchronous call to a procedure reqs (line 9), that processes the request list. The reqs procedure checks if r is empty, and if so, reschedules itself by *asynchronously* calling itself. If instead, the list is not empty, it allocates memory for the first request on the list, makes an asynchronous call to client (line 16), which handles the request, and then (synchronously) calls itself (line 18) after moving r to the rest of the list. The procedure client handles individual requests. It takes as input the formal c which is a pointer to a client_t structure. In the second line of client the pointer c is dereferenced, and so it is critical that when client begins executing, c is not null. This is ensured by the check performed in reqs before making the asynchronous call to client. However, we cannot deduce this by treating asynchronous calls as synchronous calls (and using a standard interprocedural dataflow analysis) as that would additionally conclude the unsound deduction that r is also not null when client is called.

The work tackles this unsoundness of applying synchronous only interprocedural analyses to asynchronous programs. It creates an *asynchronous control flow graph* for the whole program and defines a special node called as the *dispatch node* modeling the asynchronous call dispatcher in the *asynchronous control flow graph* of the program. The analysis records the *pending* asynchronous calls with local and global data values at the call location as data flow facts. The work then presents an algorithm to precisely calculate the *meet over all valid paths* in the asynchronous control flow graph for the program. The algorithm converts an AIFDS instance into an IFDS instance over the lattice of cross products of *number of pending asynchronous calls* and the local data flow facts at the call points. Since the set of pending asynchronous calls with local data flow facts is potentially infinite, IFDS is not directly applicable. The approach abstractly counts the number of such calls and then calculates *over* and *under-approximate* solutions for the exact AIFDS approach. They prove that the approach is guaranteed to converge to the exact solution.

Our static analysis implementation in "asynchrony-aware static analysis work for Android applications" is built upon this static analysis work. We extend the AIFDS approach to Android applications by correctly modeling the asynchronous control flow semantics in Android Inter-

```
1   global request_list *r;
2   main(){
3   ...//setup request list r
4   async reqs();
5   ...//dispatch loop
6   }
7   reqs (){
8     if(r == NULL){
9       async reqs ();
10      return;
11    }
12    rc = malloc (...);
13    if (rc == NULL){
14      return ABORT;
15    }
16    async client (rc, r -> id);
17    r = r -> next;
18    reqs();
19  }
20  client (clien_t *c , int id){
21    ...
22    c -> id = id;
23    ...// comtinue processing
24    return;
25
26  }
```

Figure 3.4: Example : Asynchronous program from [68]

Component Control Flow Graph (AICCFG) and further extending the Asynchronous Data Flow Algorithm (ADFA) [68] to handle Android component lifecycle semantics and *inter-component communications*. For instance, Android ICC calls have complex runtime semantics and lack explicit asynchronous calls and returns. The asynchronous calls are either due to ICC or callbacks from the framework to the application. Moreover, these are asynchronous calls to a collection of callback methods rather than a single target method, which needs to be resolved either explicitly or by using the manifest. Once the target is resolved, all the valid paths should be invoked based on the called component type and its lifecycle. Chapter 4 discusses the details of our analysis.

3.4 Static Typestate Analysis

3.4.1 Classical Typestate Analyses

3.4.1.1 Original Typestates

Typestate tracking or static typestate analysis was first defined in the seminal work by Strom and Yemini [115]. Since then, there have been various attempts to statically verify or check typestate properties of programs from different domains. Since typestate tracking requires tracking temporal safety properties of a program, it requires sound (refer Chapter 4 for the

definition of soundness) tracking of both control and data flow of the program. Moreover, for practical purposes, the approach should be substantially precise. This work presented the concept of *typestates* as an extension to the concept of *types*. It aimed to find nonsensical program executions in programs, like using a variable without prior definition, reading a closed file, etc. This is achieved by associating a *states* set with each *type* in the program. This set is defined as *typestates*, and each procedure of the language has a *Pre* and a *Post* typestate set which captures possible typestate automata transitions. The work presents a *two pass typestate tracking* algorithm which takes a *program graph* without typestate labels, and it inserts typestate labels and *coercions*. It produces a typestate consistent program graph if the program does not violate the related typestate property. The work also discusses how typestates affect various features of a programming language by taking an example of a small programming language **NIL** [116]. The work discusses the challenges associated with tracking typestates in real-world programs with rich programming features like Pointers and aliasing but abstracts away these features in NIL. Most of the real world imperative programming languages like Java, C or C++, extensively support pointers or references and aliasing, and this is a major hurdle in precisely tracking typestate properties of these programs using the approach described in the paper. Besides the problem of aliasing and dynamic memory allocation, the original typestates can only express finite state abstraction properties. Unfortunately, many of the real world programs require checking much richer properties which are beyond regular languages. The first limitation is addressed by various works which present typestate tracking algorithms and approaches for programs in the presence of aliasing [53], but these works are still limited in scope because they apply the known IFDS approaches to typestate analysis which are both imprecise and unsound for programs with richer features. The second limitation is still unresolved and we present a generalized version of typestates which can express and verify such richer program properties.

3.4.1.2 Typestate Analysis in the Presence of Aliasing

Fink et. al. [53] present a verification technique for typestate properties in real-world Java programs. The work presents a range of typestate verification techniques with varying degrees of precision and uses these for a staged typestate verification. The work presents a typestate analysis framework which allows one to formally define different levels of abstraction and precision. It presents an integrated typestate and alias analysis built over this framework. The work is related to typestate tracking work for objects [82], but differs from our work on various counts, the most significant one being the ability to deal with non-regular program properties. Many of the approaches to handle aliases in this work could be useful in our typestate system

as well and we discuss briefly one of these in chapter 5.

3.4.2 Typestate Analysis for Android

Android application framework exposes a large set of APIs for Android applications to interact with the device hardware, resources and other applications. Most of these API's have an associated usage constraints or protocols. Most of the static analysis research for Android applications is directed towards *information flow* and other security relevant problems. This may be attributed to the difficult challenges associated with precise tracking of typestate properties in the case of Android applications. Ours is the first static typestate analysis work for Android applications. The details are discussed in Chapter 4.

3.5 Language support for Typestate

3.5.1 Typestate for Objects

Typestates are particularly useful in capturing state based properties in imperative object-oriented languages where the states of objects mutate over time. In "Typestate for Objects" [82], Deline et al. present a sound, static typestate system for imperative object-oriented programs. They present a modular typestate system soundly capturing the typestate change semantics due to inheritance in object-oriented programs. The work contributes to typestate analyses in distinct ways:

first, the work presents a modular typestate system (a type system for typestate). This typestate system captures object's state as a function of contents of the fields of an object. At any time the object can be typed either as a superclass, the current class or any of its subclasses. Statically, the type checker can only know about the fields of the superclass or the object's declared class. Thus, some object states introduced by its subclasses are unknown. Typestates are good for capturing such program states in a modular fashion since they abstract away the details of subclass fields and only keep track of object's state by static names. The approach allows a clear description of how subclasses can extend the interpretation of typestates and language features like upcast, downcast, etc. To achieve this, the work defines typestates associated with an object as a set of *frame typestate*, one for each class frame of the object's dynamic type. It provides a uniform abstract frame for each unknown subclass. For example, the work defines an object typestate σ_C as follows:

$$
\begin{array}{rrcl}
\text{(Object typestate)} & \sigma_C & := & \chi_C :: \text{rest @s} \mid \chi_C :: \bullet \\
\text{(frames)} & \chi_C & := & \chi_B :: \text{C @ s} \;\; \text{where B} = \mathsf{baseClass(C)} \\
 & & & \mid \text{Object@s where C} = \mathsf{Object}
\end{array}
$$

The above definition states that, an *object typestate* σ_C is a collection of *frame typestates* χ_C and either a **rest** typestate s (specifying that all possible subclass frames of C satisfy s), or no rest state indicating that the dynamic object type is exactly C (for example, right after new, or because class C is restricted from having subclasses, as with abstract classes in Java]). A collection of frame typestates χ_C consists of a frame typestate for frame C and each supertype of C. Class C in an object typestate σ_C corresponds to the traditional static type of an object.

Besides a modular typestate description for objects, the work also presents typestates as a generalization of object invariants. It looks typestates as a different invariant of an object over its lifetime and defining typestate changes via annotating each method with Pre- and Post-typestates. Further, the typestate system allows transitioning object invariants and incremental state changes. The work also provides a small, imperative, object-oriented core language with a static typestate system implementing the modular typestaes of the work.

The typestate system discussed in this work suffers from expressive limitations common to other regular typestate works. It can only express *regular* typestate properties. Moreover, the work does not provide any concrete implementation of their theoretical typestate system. Our p-typestates work tries to address some of these limitations by giving a more expressive notion of typestates, a richer type system and a concrete implementation of a typestate-oriented language with the p-typestate type system. It is important to note here that our p-typestate definition is not modular in the exact sense of object frames as discussed in this work.

3.5.2 Typestate oriented programming

Typestate oriented programming [9, 31], introduces concept by formalizing a nominal object-oriented language with mutable states, that integrates typestate change and typestate checking as primitive language concepts. Most of the other imperative languages like Java, C++, Python, etc. cannot express typestates directly. Typestates and typestates tracking are encoded through a disciplined use of member variables or fields. For instance, consider the *FileManager* example discussed earlier. The state of the file can be encoded in a field of a File object, which can be null for *closed* file and *non-null* for *open* ones. Such an encoding of typestates makes it difficult to comprehend or debug these programs. Comprehension is hampered because the protocols underlying the typestate properties, which reflect a programmers intent, are at best described in the documentation of the code. Also, such typestate encodings cannot guarantee by construction that a program does not perform illegal operations. Checking typestate violations can be done through a whole program analysis [53], or with a modular checker based on additional program annotations [18]. In either case, the lack of integration with the programming language hinders adoption by programmers.

Although, there are a few indirect ways to encode typestates and typestate transitions via *types* (by allowing type associated with a term to change during its life), using standard type systems(which remains constant during object life), we cannot naturally specify and verify dynamic behaviors of the program described by typestates. On the other hand, providing the programmer with a rich logic and a language for writing pre- and postconditions quickly becomes undecidable. For instance, the ESC/Java System [56] allows rich specifications to model dynamic typestates like properties but has an undecidable static type checking in general.

Garcia et al [9] presents the foundational concepts for a typestate-oriented programming language which directly supports typestates and typestate transitions. The language design incorporates typestates as first-class objects. It defines a statically typed core language called Feather Weight Typestate (FT) inspired by a core object-oriented language called Feather Weight Java(FWJ). The work further presents a gradually typed extension and practical implementation of FT called as Plaid. Plaid [31] language allows to define mutable *states* similar to *classes* in a normal object-oriented language and has both static and gradual typestate checking. Although, in our experience, Plaid predominantly checks typestate properties at runtime. Each method has pre- and post- typestate annotations which capture typestate transitions and can be verified statically or dynamically. For example, the following signature for a method m, describes typestate transitions for fields f and g.

$$void\ m(OpenFile \gg CloseFile\ f, OpenFile \gg OpenFile\ g)\{...\}$$

Figure 3.5 presents a code fragment for a small logging example, the code is adapted from [9]. The OpenFileLogger (OFL) state holds a reference to a file object (OF) and provides a method (log) for logging to the file object. When the logging is complete, the close method closes the file and transitions the state of the base object from OFL to FL. Plaid provides syntax to annotate pre- and post- states showing possible typestate transitions associated with a method. For example, consider the close method in the figure. It requires the pre- state of the FileLogger object to be *opne* (OpenFileLogger) and updates the post state to *not open* (FileLogger). This transition is shown as close () [OFL $\gg$ FL]. The client code, has a staticLog method which requires an *open* FileLogger, and finally, the logger is closed. The states of a typestate property automaton are represented as Types (a *state* is a type with the same name). These and other features of Plaid [31] allow one to model and verify useful typestate properties directly in the program. One particular limitation of Plaid's typestate and the type system is, that it can only express regular program properties. Thus, we can easily express and verify a property like the FileLogger but cannot express a non-regular property like, "Number of *read* operations

```
1  class FileLogger { /* FL Logging related data and methods */ }
2
3   class OpenFileLogger : FileLogger { /* OFL */
4     OF file;
5     void log(string s)[OFL] {...}
6
7     void close()[OFL ≫ FL] {
8       full(OF) OF fileT = (this.file :=: new OF("/dev/null"));
9       fileT.close();
10      this ← FileLogger();
11    }
12 }
13
14  // Client code
15  void staticLog(OFL logger) {
16   logger.log("in staticLog");
17  }
18
19  OF file0 = new OF(...);
20  OFL logger = new OFL(file0);
21
22  staticLog(logger);
23  logger.close();
24 }
```

Figure 3.5: An example program in Plaid

to a file are greater than the number of *write* operations ". The p-typestate work presented in Chapter 5 particularly addresses this limitation of regular typestates and typestate-oriented programming languages like Plaid, Fugue [82], Plural [18], Hanoi [89], and frameworks for typestate analysis [20]. Our work builds upon the implementation of Plaid. We could have implemented our typestate system over a more basic core language like FWJ equally well, but since Plaid is specifically tied to typestates, using Plaid as base explicitly allows us to present the expressiveness of our richer p-typestates.

Next, we discuss some of the other works for modeling and verifying typestate properties. Each of these works suffers from similar expressive limitations of the typestate and the type-checking system as are being suffered by Plaid and "typestate for objects" [82].

3.5.2.1 Plural

Plural [18] from Aldrich et al. provides another typestate-oriented type system and an automated static analysis to modularly analyze various typestate properties of programs. This is an extension to the object-oriented paradigm where the objects are modeled in terms of their *states* and state changes along with the classes. The Plural system extends the earlier mentioned Deline et al. work on "typestates for objects" [82]. Plural uses the *Java Specification Language* to annotate methods and fields with pre and post states (using annotations like *guarantee*, *requires* and *ensures*). These annotations allow checking each method separately and inductively

build a modular analysis for the overall program. Thus, effectively, Plural's annotation system and the permission system is closely related to the Plaid's typing annotations. However, the Plural system does not alter the runtime representation of classes in any way and the analysis is purely static, while Plaid provides gradual typing and dynamic state changes for objects. The Plural's annotation system suffers from expressive limitations similar to that of Plaid, and cannot express non-regular typestate properties.

3.5.2.2 Clara

Clara [20] by Eric Bodden is a framework for implementing *hybrid* (static and dynamic) typestate analyses for Java programs. The framework decouples runtime monitoring from the static analysis components. The user of the framework needs to define a set of runtime monitors for the typestate property being analyzed. This is generally done using a runtime monitoring tool, which generally generates simple *AspectJ* [124] aspects. The tool also needs to generate special annotations to the aspect which are used by the static analysis component to *invoke* or *disable*. The generated aspects are then weaved by CLARA into the program while the static analysis component analyzes and invokes these runtime checks at instrumentation locations which are relevant to the typestate property being checked and which are not being verified by the static analysis. This gives CLARA an optimized instrumented program that updates the runtime monitor only at locations that remain enabled.

Besides these works, there are a few other works [89, 96] on typestate verification and other aspects of typestates. The major difference between any of these regular typestate works and our parameterized typestates work lies in the expressiveness of the system and purely static verification. It will be interesting to apply the concepts discussed in some of these works, like gradual typechecking to p-typestates. This may allow us to express richer properties and reduce the annotation burden from the programmer but will increase the runtime cost of the programs. We leave such extensions as possible future work.

3.5.3 Extended Static Checking for Java

Extended Static Checking (ESC) [56], has the common goal of lightweight programming verification much like typestates. The work provides programmers with a property specification language and performs an extended static checking of these properties, in the sense that it allows specifying and verifying richer properties than normal type systems and type checkers. The static checker uses verification condition generators and automatic theorem provers. The work allows annotating methods and field declarations with automatically checkable annotations. For instance, a requirement that a field *input* should not be *null* in a method body can

be represented via an annotation like:

$$//@requires\ input\ \neq null$$

The annotation language also allows to specify richer constraints and annotations like:

$$//@invariant 0 \leq size, size \leq elements.length$$

ESC [56] gives an expressive static checking mechanism, but the expressiveness comes at the cost of decidability of static checking. The static checking system of ESC is undecidable. Further, the static checking discussed in ESC is unsound in the presence of loops, unless the loop invariants are provided by the programmer. Contrary to this, our aim is to have a static type system which is sufficiently expressive to verify important non-trivial program properties and yet has a decidable typechecking and inference (in practical cases). The details of how our work overcomes some of these limitations are presented in Chapter 5.

3.6 Fully Dependently Typed Languages

Coq, Agda, Caynene [117, 98, 14], etc. are languages or theorem provers with *full dependent types* support along with *parametric* types. This means they fully capture Per Martin Löf's type theory [22]. Availability of dependent types makes these languages highly expressive as dependent types can also be used to represent predicate logic. Under the interpretation, called "propositions as types" (refer Chapter 2), the operations on dependent types relate to quantifiers in predicate logic. For instance, if $\phi(a)$ is a proposition dependent on $a \in A$, then an element of the dependent product $\Pi_{a \in A}\phi(a)$ consists of a function assigning, to each $a \in A$ an element of $\phi(a)$, i.e. an assertion that $\phi(a)$ is true. This allows us to specify and prove any predicate logical properties in these languages. For instance, dependent types allow types encoding of a sized list with integer length as the dependent index. Thus we can define types such as *List(4)*, a *List* with size 4 or types like *Tree(n)*, a *Tree* of height *n*. Dependent function types further allow to define a refinement of these types [112, 127], defining some invariants over the indexes capturing the length (or other similar properties) of the *data*. For example, using dependent types we can eliminate the need for runtime checks for array-bounds, enforcing the invariant over the index i being accessed in an array and the *size* of the array:

$$get : \Pi(size : nat).\Pi i\{i : int, 0 \leq i \leq size\}.Int(i) \rightarrow List(size) \rightarrow element$$

Listing 3.1 presents a code fragment showing a *SizedList* in the syntax of our language. It uses dependent types (from the p-typestate type system) to define a List with a dependent type SizedListTy parameterized with integer n and a normal List. The list has two fields: a head of the type SizedCons, which is a constructor for a sized element and a tail of the type SizedList. The size information associated with the List allows definition of statically verified basic methods of SizedList. For example, the prepend method takes a List element elem as a parameter and updates the List size from n to $(n + 1)$ by adding the elem to the head of the SizedList. The more interesting method is the get method. It takes a BoundedInteger type index (a BoundedInteger is a type for an integer with bounds). The method is undefined for indicies greater than or equal to the size of the SizedList n and returns the element at the index otherwise.

Listing 3.1: An example of SizedList using dependent types in our language syntax

```
1   package plaid.iisc.lang;
2
3  /* @author- Ashish:
4  * @description - SizedList - A List with size information in the type.
5  */
6  import plaid.iisc.lang.testing.test;
7
8  // head : ListCell , tail : List
9  state SizedList case of List{
10         type SizedListTy : Pi (n) -> List;
11         var SizedCons head;
12         var SizedList tail;
13
14         method void prepend(elem)[unique SizeListTy(n) -> List >> unique SizedListTy(n+1) -> List]{
15                 this.head = new SizedCons {var value = elem; var SizedCons next = this.head;};
16                 this <- SizedListTy(n+1) -> List;
17         }
18
19         method void add(elem)[unique SizedListTy(n) -> List >> unique SizedListTy(n+1) -> List]{
20                 this.tail = new SizedList {var head = new elem; var tail = new plaid.lang.NIL;};
21                 this.tail = new Cons {var value = elem; var next = new plaid.iisc.lang.NIl;}; //
                       define a  size aware cons
22                 this <- SizedListTy(n+1) -> List;
23         }
24         method void append(unique SizedListTy(m) -> List list)[unique SizedListTy(n) -> List >>
               unique SizedListTy(n+m) -> List]{
25                 match (list.tail){
26                         case Nil{
27                         this.tail = this.tail.add(list.head);
28                         this <- SizedList(n+1) -> List;
29                         }
30                         case Cons{
31
```

```
32                      this.tail = this.tail.append(new Cons {var value = list.head.value; var next
                            = list.tail;});
33                      this <- SizedList(n+m) -> List;
34                      }
35                      default{
36                      java.lang.System.out.println("bad");
37                      }
38
39              };
40
41
42
43      }
44      method get(unique BoundedInteger(i, i < n) -> plaid.iisc.lang.math.Integer index)[unique
            SizedListTy(n) -> List]{
45
46              var returnValue = new plaid.iisc.lang.Nil;
47              var counter = 0;
48              this.map(fn (x) => { /* map is a normal map function defined over List */
49                      if(counter == index){
50                              returnValue = x;
51                      };
52                      counter = counter + 1;
53              });
54              returnValue;
55
56      }
57      method reverse()[unique SizedListTy (n) -> List >> unique SizedListTy (n) -> List]{
58              match (this){
59                      case Nil{
60                            this;
61                      }
62                      case Cons{
63                              new Cons{var value = this.tail.reverse(); var next = this.tail;};
64                      }
65                      default{
66                              this;
67                      }
68              };
69      }
70      method map(f){
71              new SizedList {var head = new Cons{val value = f(this.head.value);
72              var next = this.tail;}; var tail = this.mapHelper(f, this.tail);};
73      }
74      method mapHelper(f, list){ // add the support for typed variable like (SizedList list)
75              match (list.head){
76                      case Cons{
77                              val newHeadValue = f(list.head.value);
78                              new SizedList { var head = new Cons {var value = newHeadValue; var
                                    next = list.tail.head;}; var tail=this.mapHelper(f, list.tail)
                                    ;};
79                      }
80                      case Nil{
```

```
81
82                                      list;
83                              }
84                               default{
85                                      java.lang.System.out.println("bad");
86                              }
87                      };
88              }
89      }
```

A fully dependent typed language can have complex indexes for types like $\Pi(x = get(i)).T(x)$, which takes the return value of the *get()* method as an index for another type $T(x)$. Unfortunately, this expressiveness comes at the cost of decidability of type checking, making it undecidable for general dependent type theory. Thus, these languages or proof systems are not well suited for fully automatic typechecking of rich program properties. Moreover, the type theories have semi-automatic verification, where the burden of the typestate checking or property checking is on the programmer, as she needs to provide a proof of a property. A possible approach is to restrict the dependent terms of these languages to belong to a decidable theory. This is semantically equivalent to defining a language equivalent to ours (minus the p-typestate features) in these fully dependently typed languages like Coq or Agda. Further, to achieve this, a programmer needs to build a Presburger Theory in Coq, restrict the dependent terms to this theory and then write properties and programs in this restricted sub-language. This encoding will be verbose and complex for a programmer to build and will further require a proof of decidability of typecheking. Moreover, it will still lack the expressiveness to define rich p-typestate like pre- and post- conditions and p-typestate changes.

3.7 Dependently Typed Language Extensions for Regular Types

Many useful program properties require the expressiveness of dependent type theories while at the same time they require decidable typechecking and typeinference. This has motivated various works which extend a language (both from the imperative and the functional domain) with normal types to include *dependent types*. These languages use the power of dependent types to define invariants associated with data in its associated type but rather than choosing the full power of dependent types, they restrict the index language of the dependent product or sum types to a decidable logical family. This gives a sufficiently expressive type system without losing the decidability of the type checking. Here we discuss a few dependently typed language extensions which are most closely related to our dependently typed core typestate-oriented language and how they differ from our work.

3.7.1 DML and Other Refinement Types

Refinement Types are a variant of dependent types where a type has an associated *refinement*. A refinement can be any constraint or predicate over the auxiliary variables defining some dynamic property of the data. For example, (int (ν) $\{0 \leq \nu \leq n\}$), represents a refinement *int* type with a default auxiliary variable ν. The type represents an *int* with value in between 0 and n. Using such a syntax, a refinement type allows defining rich invariants over data type. Liquid types [112], is a refinement type which allows the refinement to be a logical formula, hence the name *logically qualified types*. The seminal work in refinement types is a dependently typed extension of ML [91] call Dependent ML or DML [127]. Our core language is also inspired by DML and hence there are several similarities between these other refinement types and our language. However, our work differs from these in terms of being dependent extension of typestates, rather than of simple types (type associated with a data object remains constant throughout its life, while typestate is allowed to transit from pre- to post-condition), the language domain, decidability of type-checking, handling of loops and recursive data structures, etc. For example, DML adds dependent types to ML type system. It is a functional programming language, while our core language is an imperative, typestate-oriented dependently typed language. Our loop invariant calculation technique can be useful for type-checking general recursive functions in DML.

3.7.2 Xanadu

Xanadu [126] is an imperative version of the DML described earlier. The work although is not related to typestates in particular but it allows the types of variables to change during evaluation thus relating it to typestate to some extent. Xanadu provides a static dependent type system and aims at eliminating array bound checks in imperative programs. Although closely related, the work differs from our p-typesates work on various accounts. Although Xanadu allows types to change during evaluation, it provides no support for specifying or verifying pre- and postconditions of methods, expressions and arguments which forms the basis of typesate like properties. For instance, it is hard and unnatural to model a simple FileManager protocol requiring file being read, only in an open state. Properties defined and verified by Xanadu can be easily specified and verified as p-typestate properties in our work. The loop invariant calculation approach presented by us is significantly different than their approach and obviates the need of loop invariant annotations from the programmer. Compared to this the "Master type" based "state type" calculation for loops in Xanadu requires annotation in certain cases or can be highly imprecise, otherwise. Besides this, we present a state/object-oriented language

with specific issues related to subtyping, aliasing, etc. These concerns are further important for correct tracking of typestate properties, which Xanadu does not need to be concerned about.

3.7.3 X10

The constrained type of X10 [99] is related to the dependent type language we provide in our work. X10 allows defining "constrained types" which are dependent types with logical expressions over properties, final instance fields of a class, and final variables, in the scope of the type as dependent terms. It also allows different constraint systems as compiler plugins. Since both X10 and BR-Typestate are related to DML, there are a few similarities, yet some important differences.

Firstly, the major focus of our work is to attack the expressive limitations of typestates, thereby making it possible to verify non-regular program properties and protocols, while X10 is targeted towards static checking of generic constraints.

Secondly, we have a separate index definition language and index terms are separate from the class fields and other variables. This indexing language is simple and contains constraints over integer variables, this limits the expressiveness of our type system as compared to X10's constrained types but gives us a decidable type-checking without the need of a symbolic checking as is needed in X10.

For instance, X10 type may contain a final field or even a method calls as an index of a type, this allows to give far more expressive constraints but requires a symbolic execution to get the possible properties of these fields and method executions. The symbolic execution could lead to an undecidable type-checking in general.

Finally, X10's type-system allows the conditional expressions to be a conjunction of different constraint families, thereby making it highly expressive but lacks a formal study of the decidability of the type-checking over these complex expressions.

3.8 Loop Invariant Calculation

There is a substantial amount of work in the field of automated loop invariant inference, spawning more than three decades. This is an evidence of the role, loop invariants play in automated verification of programs. This section discusses some of the related works in the field without the pretence of being exhaustive. Our work clearly falls into the category of white-box, static technique of loop invariants calculation and is a novel approach to compute loop invariants. The work most closely related to our work is *accelerating invariants generation* by Kumar et al. [81]. We discuss this work in detail after discussing some of the more general approaches to synthesize loop invariants.

3.8.1 White-box Techniques for Loop Invariant Calculation

As discussed in the Chapter 2, there are various techniques under white-box methods and all these techniques/work are aware of the program and the properties being verified. The white-box technique may further be classified into, either *static* or *dynamic*. Static techniques for loop-invariant calculation are more common and more recently, that dynamic techniques are also being applied successfully to the problem.

3.8.1.1 Static Techniques

Abstract Interpretation and *constraint based* approaches are the most common static techniques for static invariant inference. Jhala et *al.* [68] provide an overview of some of the important static techniques and how they are useful. *Abstract Interpretation* performs a symbolic execution over the abstract domain of the program variables to verify or check some property. The seminal work in this domain came from Cousot and Cousot [39] and the technique and was later extended to handle features of modern programming languages like memory-management and objects [78, 30]. *Constraint-based* methods rely on sophisticated decision procedures over non-trivial mathematical domains like convex polyhedra. These domains are used to represent the semantics of the loop with respect to some property of interest. Our work using *acceleration* techniques (refer Chapter 5) belongs to this methodology of calculating invariants. One important property of static methods is that they are sound and often complete with respect to the class of invariants that they can infer. Soundness and completeness are achieved by using decidable Techniques in the underlying mathematical domains they represent. This implies that the extension of these techniques to new classes of properties is often limited by undecidability. State-of-the-art static techniques can infer invariants in the form of mathematical domains such as linear inequalities [40], polynomials [113, 111], restricted properties of arrays [24, 23, 64], and linear arithmetic with uninterpreted functions [16]. Static loop invariant calculation techniques can benefit from annotations in the form of method specifications or some loop invariants. A few of the static techniques use these annotations [102, 73, 70].

Lahiri et al. [73] also leverage specifications to derive intermediate assertions, but focusing on lower-level and type-like properties of pointers. [102] derive candidate invariants from postconditions within a framework for symbolic execution and model-checking. [70] derive complex loop invariants by first encoding the loop semantics as recurring relations and then instructing a rewrite-based theorem prover to try and remove the dependency on the iterator variables in the relations. The survey work from Furia et al. [59] provides a detailed list of loop-invariants for some important algorithms and other related works and how these works evolved. Reader should refer to [59] for further details.

3.8.1.2 Dynamic Techniques

Ernst et al. [47] presented the first dynamic approach for invariant inference. It gave rise to many derived works like [104, 41, 107, 62]. The basic idea of dynamic invariant inference technique is to test a large number of candidate properties over a large number of program runs. The properties which are never violated are more likely to be *invariants*. This suffers from the general unsoundness of dynamic approaches. Nonetheless, it is quite effective and useful in practice.

3.8.2 Black-box Technique for Loop Invariant Calculation

There are a few works using learning techniques which are agnostic about the program and the properties being verified [48, 55]. Cobleigh et al. [36] presented learning in the context of invariant calculations for the first time. This was followed by applications of the L^* algorithm [10] to find rely-guarantee contracts. Houdini [55] uses conjunctive Boolean learning to learn conjunctive invariants over templates of atomic formulas. Recently, there has been an increased interest in applying *scalable machine learning techniques* to an invariant calculation. They use machine learning techniques to find classifiers that can separate positive examples and good states from the counter examples [129].

In the same domain of learning based black-box technique of loop invariant calculation, Garg et al. [60] present a learning framework for synthesizing invariants. The ICE-framework has two main components, a *teacher* and a *learner* which iteratively interact to find the invariant.

3.8.3 Other techniques

The work most closely related to the loop invariant calculation approach presented by us is by Madhukar et al. [81]. The work provides a detailed experimental study to show that *accelerators* can support/aid program analyzers to improve the invariant synthesis. The work uses two main analyzers with limited invariant calculation abilities, the CMBC [34] and the IMPARA [71], and two further tools with a broad range of loop invariant calculation techniques. Although the work empirically shows the correlation and possible applicability of *acceleration*, it does not use acceleration as a loop invariant calculation technique. It instruments the loops in the program with its *closed accelerated form* and then calls off-the-shelf analyzers. Compared to this, our approach uses the exact set of reachable states (REACH) for Presburger-definable loops and either uses this exact formula or some derived form of this as an invariant for the loop. This work performs loop acceleration instrumentation over C programs using the acceleration technique, while our approach can work over any program with Presburger definable loops (loop guards and expressions in the body). Further, the emphasis of our work is to use the acceleration output

for a restricted class of systems (Presburger-definable falttable counter systems) as a possible invariant and inductively verify a given p-typestate property while their work emphasizes on empirically showing that accelerated loops aid analyzers in verifying properties compared to non-accelerated loops.

3.9 Chapter Summary

In this chapter, we presented a detailed survey of existing works related to our contributions in this thesis. There are two major sections in the chapter. In the first section, we presented the existing works related to Android formal modeling, asynchronous static analysis for Android, existing typestates analysis for Android, etc. The second section contained related works for the Parameterized typestate work discussed in Chapter 5. This contained existing related works for typestate-oriented programming, regular typestates, other fully dependent typed languages, dependently typed languages extension of simply typed languages, etc. Finally, we presented existing works, related to our loop-invariant calculation approach.

Chapter 4

Asynchrony-aware Static Analysis of Android Applications

4.1 Introduction

Android is the heaviest used and rapidly growing mobile system with more than 2 million paid and free applications on Google's Play store currently [3]. Many of these applications are rigged with benign and harmful bugs and vulnerabilities. Unfortunately, analyzing these applications is a herculean task. There are numerous static [76, 13, 125, 79] and dynamic program analysis [46] works which try to find bugs in Android applications.

Android applications are typically made up of four types of elements called *components*. These are *Activity, Service, BroadcastReceiver, ContentProvider*. Each of these components has sets of methods or callbacks which are called by the Android framework on various user or system events. This differentiates these applications from normal Java programs and makes the analysis of these applications challenging. The control and data flow in Android applications are further complicated due to extensive use of *asynchronous calls*. These calls make it convenient and efficient to model events and the interactions between various *components*, and permit the Android framework to interleave the execution of several event handlers and other callback methods.

Although many of the static analysis works [76, 125, 13, 79] for Android applications model these interactions with the framework, none of these works correctly model the asynchronous behaviour of the event handling mechanism and inter-component communications in Android. Moreover, they also lack a correct modeling of the framework's interaction with the application components (called *component lifecycle*). These limitations bring unsoundness and imprecision in these analyses.

In this thesis, we present the first formal modelling of asynchronous control flow semantics in Android applications and a sound (single threaded non-preemptive control flow as described later in Section 4.5), and a precise model and semantics for *component lifecycle* of Android applications. We explicitly model all the asynchronous calls and framework callbacks by creating an *application environment model* for the application and create a precise lifecycle state machine for each component. We define an Android inter-component control flow graph (AICCFG) which integrates these two and soundly models all possible control flow interactions between various units.

We present an algorithm to generate such an AICCFG for the application. We implement and solve a client typestate [115] analysis to verify Android resource API usages as an instance of the asynchronous interprocedural finite distributive subset (AIFDS) problem [68] and compare the results of our analysis against an asynchrony-unaware version of the analysis on the program representation used by other state-of-the-art analyses for Android applications. We demonstrate empirically that these asynchrony-unaware techniques are both unsound and imprecise with respect to our asynchrony-aware analysis. We present a set of 19 benchmark applications in five different resource categories, called AsyncBench. This comprises of test applications that use various resources in both safe and unsafe manner. A sound verification of these test applications requires an asynchrony-aware modeling of the applications. We were able to verify all the typestate violations in these applications with a precision of 78% and recall of 100%. The comparison of these results against the asynchrony-unaware analysis over unsound program representation used by other works clearly demonstrates the soundness and effectiveness of our application modeling and analysis.

The major contributions of our work are as follows-

- We present the first sound model of asynchronous control flow and component lifecycle semantics in Android applications.
- We present an intermediate representation of Android applications called AICCFG based on the above model.
- We develop a typestate analysis over AICCFG and find resource API violations for a variety of resource types.
- We present a set of benchmark applications called AsyncBench, comprising of applications whose analysis requires a sound modeling of asynchronous and lifecycle semantics of Android applications.

- We finally compare our typestate analysis against an asynchrony-unaware analysis and demonstrate the benefits of our modeling and analysis against these.

4.2 Motivating Example

4.2.1 Control Flow in a Typical Java Program

Consider a simple Java program in Figure 4.1 with a bunch of method calls. The control flow for this program is shown in Figure 4.2. Such a graph captures all possible control flow paths for any execution of the program. The control flow graph for a singe procedure or function is called an *intra-procedural control flow graph*, while the one capturing control flows across procedures is called an *inter-procedural control flow graph*. An inter-procedural control flow graph is an input to many of the static program analysis algorithms and tools [110, 68, 119]. Unfortunately, an Android application is not a simple Java application as discussed in Chapter 2 and the control flow in Android differs extensively than control flows in normal Java programs.

Next we discuss some of these differences and discuss the limitations of the state-of-the-art static analysis works for Android in correct and precise modeling of these differences. These limitations will bring out the motivation for a correct modeling of Android control flow.

```
1    import X;
2    public static void main(String[] args){
3
4        int x = 2;
5        int y = x +1;
6        int z = foo (x);
7        z = x + y + z;
8        X.abc();
9
10   }
11
12   int foo (int x){
13      return fooBar (x);
14
15   }
```

Figure 4.1: A simple Java Program

Consider the example application FileReader in Figure 4.3. The application has two activities *SelectActivity* and *ReadFileActivity*. The application allows the user to select a file and open the file in ReadFileActivity. The FileReader object, line 2, is a global static reference which is accessible through both the components. A typestate [115] analysis checks the possible runtime states of a resource object against a given typestate property finite automaton. A typestate property automaton (defined formally in Section 4.6.1), is a finite state machines with states defining a set of *states* an object or a reference can be in and transitions defining valid *methods/operations* in each of these *states*. Figure 4.4, shows such a typestate property automaton for an Android *Camera* resource.

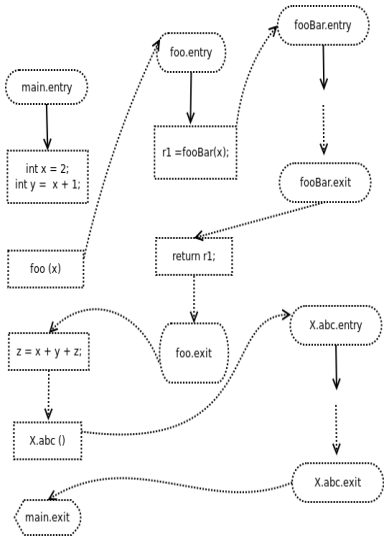


Figure 4.2: A simple cfg for the program

To understand the possible flow of control in this sample application an application developer and the analysis developer depends on official Android documentation. Unfortunately, the documentation is verbose and not easy to understand. For example, let us try to understand a simple semantics of *onPause* method completion of an Activity, from the official Android documentation-

"Completion of the onPause() method does not mean that the activity leaves the Paused state. Rather, the activity remains in this state until either the activity resumes or becomes completely invisible to the user. If the activity resumes, the system once again invokes the onResume() callback. If the activity returns from the Paused state to the Resumed state, the system keeps the Activity instance resident in memory, recalling that instance when it the system invokes onResume(). In this scenario, you dont need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. If the activity becomes completely invisible, the system calls onStop(). The next section discusses the onStop() callback." .

As evident, even a simple lifecycle rule is complex and not clear to understand, this makes understanding of lifecycle enforced control flow in Android application a difficult task for applications developers and analyses writers. This lack of comprehension manifest in programming errors for programmers and lack of soundness and precision in analyses for analyses writers. Besides, lifecycles, there are various other Android features as discussed in Chapter 2, which makes control flow comprehension and modeling a challenging task for Android applications.

For example, let us discuss about an analysis to verify a simple typestate property like, "The application never reads from a closed FileReader", in the example FileReader application,

```
class SelectActivity extends ActionBarActivity{
 public static FileReader myFileReader;
 protected void onCreate(Bundle savedInstanceState){
   ...
 setContentView(R.layout.activity_select);
 try{
   String filePath = this.getFilesDir() + '/' + "exFile.txt";
   myFileReader = new FileReader(...);
   int data = myFileReader.read();
   Intent targetIntent = new Intent(this, ReadFileActivity.class);
   // asynchronous call to the ReadFileActivity
   startActivity(targetIntent);
   }catch (FileNotFoundException e){
     e.printStackTrace();
   }
 }
 protected void onStart(){
   super.onStart();
 }

 protected void onResume(){
   super.onResume();
   Log.d(TAG, "onResume");
   try{
     myFileReader.close();
   }catch(IOException e){
     e.printStackTrace();
   }
 }
}
```

```
class ReadFileActivity extends ActionBarActivity {
 protected void onPause(){
   super.onPause();
 }
 ...
 protected void onStop(){
   super.onStop();
   try{
     ...
     int data = SelectActivity.myFileReader.read();
     Log.d("ReadFileActivity", "data " +data);
    }catch (IOException e){
     e.printStackTrace();
    }
 }
}
```

Figure 4.3: FileReader Application

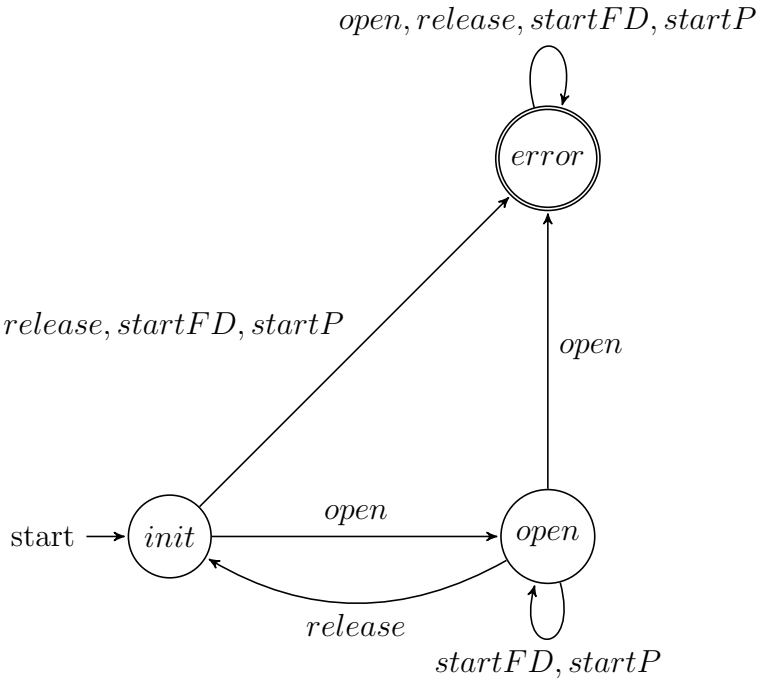


Figure 4.4: Simplified Typestate property finite automaton for Android Camera API: startFD := startFaceDetection, startP := startPreview

the analysis needs to verify and guarantee that the *FileReader.read()* is never called when *FileReader* has been closed using *FileReader.close()* and not re-opened again. Suppose such an analysis starts with the *SelectActivity*'s *onCreate()* where *FileReader* is instantiated and *File* is read at lines 8 and 9 respectively. This read operation is safe as FileReader's instantiation switches the object to *open* state. Line 12 makes an inter-component communication (ICC) call to *ReadFileActivity*. The correct semantics of Android on such an ICC call is as follows

- The ICC call is treated as an asynchronous call. An *asynchronous* call is a method call which instead of being dispatched and executed at the call site, is stored in a task queue (associated with the thread) and is dispatched for execution at some later time.
- The lifecycle callback *onCreate* has an atomic execution semantics.
- Once *onCreate* finishes execution, the Android framework (the *ActivityManagerService*) schedules onStart, onResume and onPause callbacks in that order (if present), before the control could escape to another component.
- Once *onResume* finishes execution, since the application is not overriding onPause, the call to the target Activity at line 12 is scheduled for dispatch , and ReadFileActivity's

onCreate is called.

There are three major features of the Android framework which govern the above semantics.

1. All the component callback methods have atomic execution, and they finish execution before another callback of the same or different component can be scheduled.

2. Each component has a control flow protocol (lifecycle), a set of ordering relations which governs the calling order and control flow between callbacks in and across components.

3. The ICC call like the one at line 12 is asynchronous in semantics and hence the actual dispatch of such calls is separated in time and is managed by the framework. In a typical synchronous call, the call is executed at the call site and the caller blocks itself and waits for the callee to return. Compared to this, the caller is not blocked in an asynchronous call, and the state of the system can possibly change between the time of the call and the time of dispatch of the callee for execution.

Let us try to see the possible control flow and its effect on the typestate property of the *FileReader* object in the example application. We will try to explain this in the light of our understanding of the way asynchronous calls and component callbacks are executed by the Android framework. Since the control flow spans across methods and even across different components, we refer the reader to Figure 4.6 which shows an inter-component control flow graph for the application. Although we later argue that this control flow graph is unsound, it will suffice to explain the correct possible control flow in the application. The figure only shows line numbers (as L12, L25, L40) which are crucial for the understanding of the flow of control. When a user or a system process starts the application, a new instance of Activity is created by the Android System and code block *FileReader.<init>* gets executed. Following this, the main entry component (called the LAUNCHER) SelectActivity is created by a call to *sa.onCreate(b)*. Since a lifecycle callback has atomic execution semantic, this code block is executed atomically. Due to this, the asynchronous ICC call at line L12 in this block is executed, but the call is not dispatched until the synchronous call to doSomething() is complete. Further, due to the asynchronous nature of the ICC call, the Android system stores the call in a queue to be later dispatched, rather than dispatching it right away and blocking the control.

Once the execution of *sa.onCreate* is complete, the control does not return to the Android System Service, rather the control is passed to the execution of the next lifecycle call back method *sa.onStart*. This is enforced by the lifecycle rule for an Activity. However, the control flow breaks in case of an Exception as shown in the figure. This lifecycle enforced control flow

has two main effects, firstly, again *sa.onStart* followed by *sa.onResume* and *sa.onPause* are executed before the stored ICC call to *ReadFileActivity* is actually dispatched. Secondly, when this call is dispatched, the myFileReader is already closed (at line L25), hence the state of the *FileReader* object is changed to *closed*. Once, the execution of *sa.onPause* is complete, the control is transferred back to the Android System (shown abstractly by control flow edges to Label 1.5, followed by an edge to Label 1). Now, since there is a pending asynchronous call to ReadFileActivity, its *onCreate* method is executed. This is again followed by lifecycle control flow directed flows, each executed atomically. Thus, after the completion of *rfa.onPause*, *rfa.onStop* is scheduled for execution and the file is read. Since, the FileReader object is in the state *close*, this read operation is a potential typestate violation. To correctly capture this possible property violation, a sound static analysis should correctly model this control flow semantics. If not, the analysis results will be unsound and/or imprecise. Next, we see how state-of-the-art approaches fail to soundly capture this control flow.

Unfortunately, all the state-of-the-art static analysis works for Android either do not aim at such a correct modeling or miss the correct modeling in the urge to apply the known static analyses techniques for Java programs to Android. Let us see how two of the major state-of-the-art static analysis works for Android, IccTA [76] and AmanDroid [125] will handle handle such a control flow. Figure 4.5, shows how IccTA handling a typical ICC in an application. The asynchronous ICC call at line 7 is followed by a call to a redirecting static method IpcSC.redirect0(). This is a synchronous standard call to the target component (onCreate method of the target) of the ICC call. Although this is easy to implement and allows straightforward extension of standard static analysis tools for Java to Android, this does not capture the correct asynchronous semantics of the ICC call and android component lifecycle. Figure 4.6, shows the inter-procedural control flow graph (with the system calls and services approximated) generated for the example FileReader application as generated by IccTA. The figure shows two subparts separated by a dashed vertical line-segment. The Application part contains an intra-component control flow graph for each application component and all possible synchronous control flows between them. The second part of the figure models the *Android System Service* (ActivityManagerService), with control flow edges showing invocation of different Components from the System Service. The Labels in the System Service are abstractions representing call and return points for the Components. The numbering, 1, 2, and 3 shows the ordering of the relevant events as modeled by the control flow graph generated by IccTA. The ordering depicts that, the file is opened (at L12), read (at L40) and then closed (at L25). Unfortunately, this treatment of asynchronous ICC calls as synchronous calls, and imprecise handling of lifecycle control flow leads to both unsoundness (missing of certain control flow paths possible during execution) and

```
1   class SelectActivity extends ActionBarActivity{
2    public static FileReader myFileReader;
3    protected void onCreate(Bundle savedInstanceState){
4      String filePath = this.getFilesDir() + '/' + "exFile.txt";
5      ...
6      Intent targetIntent = new Intent(this, ReadFileActivity.class);
7      startActivity(targetIntent);
8       IpcSC.redirect0(targetIntent);
9
10     doSomething();
11     doSomethingElse();
12   }
13   onStart(){console.log("Starting Activity");}
14   onResume(){console.log("Resuming Activity);}
15   void doSomething(){console.log("Doing Something"); }
16   void doSomethingElse(){ console.log("Doing Something Else"); }
17  }
18
19
20  class IpcSC {
21  static void redirect0(Intent i) {
22  ReadFileActivity rfa = new ReadFileActivity(i);
23  rfa.dummyMain();
24   }
25 }
```

Figure 4.5: Generated ICC handling code

imprecision. For example, the lifecycle model of these works captures many intra-component lifecycle control flows but misses possible control flows between lifecycle methods of two different component. Figure 4.7 shows an example of allowed and missed control flows for a sample application with two components. Figure 4.8 shows a specific example of such missing paths in the interprocedural control flow graph our example FileReader application.

As shown in graph generated by IccTA (and a similar graph is generated by other static analyses capable of handling ICCs), the call at line 12 is treated as a synchronous call and is dispatched as a blocking call at the call site. Thus the graph incorrectly models the control flow ordering 1, 2, 3 and since there is no possible control flow path for the sequence 1, 3, 2, it misses the actual control flow according to asynchronous semantics. As a result of this, the analysis misses a typestate violation at line 40.

Thus, existing static analysis works for Android applications fail to correctly model the asynchronous semantics of ICCs and handling of callbacks. They also lack precision and soundness in modeling the lifecycle semantics of components. In this thesis, we provide a formal model and semantics of Android asynchronous ICC, lifecycle and other control flow features. We also present an intermediate program representation called the Android Inter-Component Control Flow Graph (AICCFG). The AICCFG soundly captures the formal control flow semantics given by us. We show the effectiveness of our AICCFG by developing a client typestate analysis as

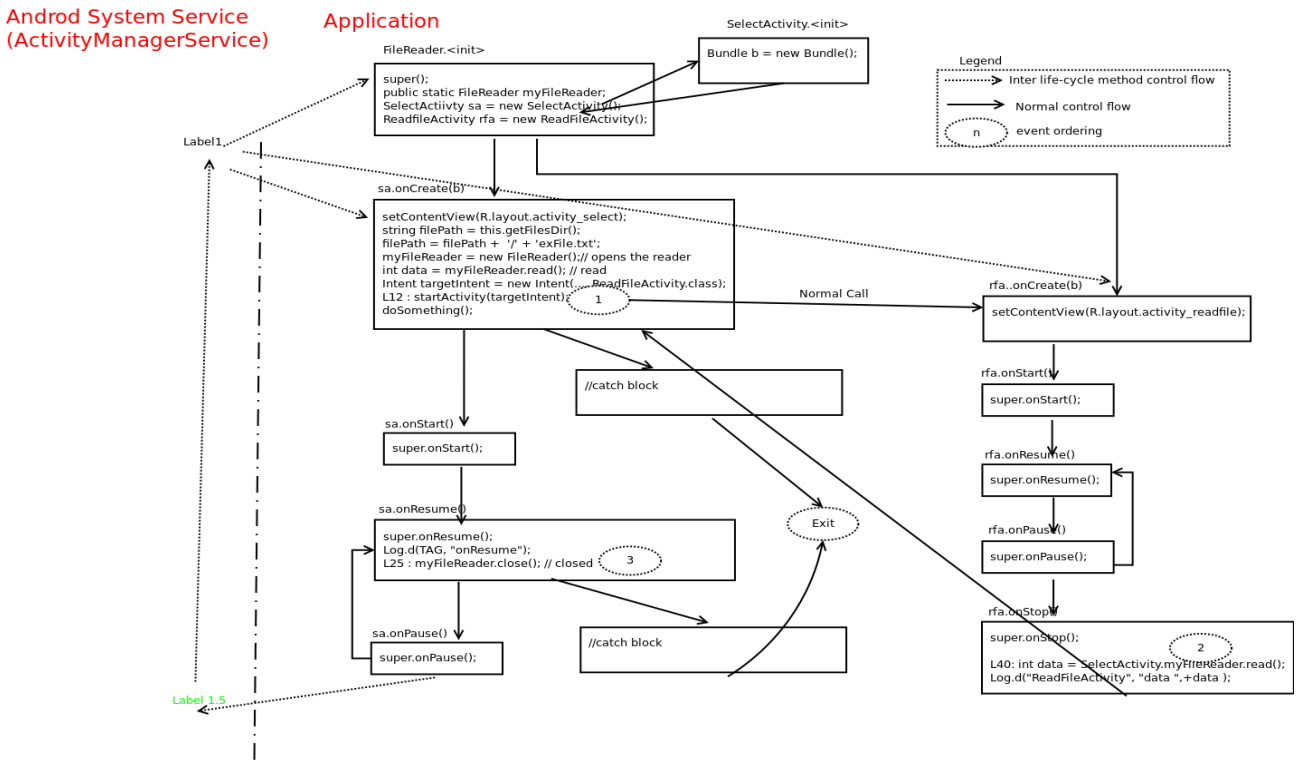


Figure 4.6: Application CFG similar to the one generated by IccTA

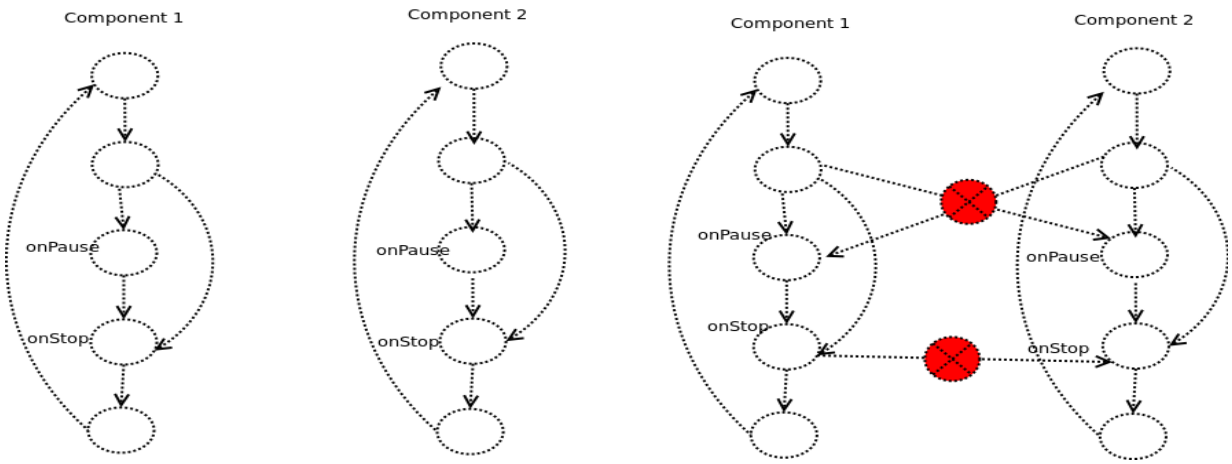


Figure 4.7: Allowed and Missing Flows in life-cycle Model

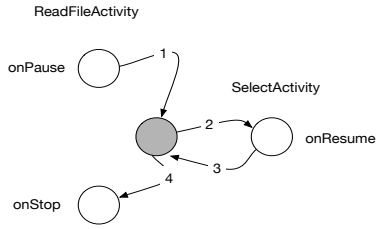


Figure 4.8: A missing ICC control flow interleaving in IccTA

an AIFDS [68] instance over AICCFG. We effectively capture several typestate violations and verify important typestate properties over a variety of applications.

Before jumping into the technical details of AICCFG definition and construction, let us see an example of the AICCFG constructed for our example FileReader application. Figure 4.9 presents the AICCFG for the application. There are three components of the graph, the left and right graph components are asynchronous control flow graphs for ReadFileActivity and SelectActivity respectively, while the *Ambiance* in the center is the model of Android system. Each square node is a basic block of instructions. For clarity, some of these blocks are annotated with the method name (like rfa.onCreate(), sa.onCreate(), etc.) when the block represents an atomically executed lifecycle callback method. There are two special kind of nodes, an *init* node for each Component control flow graph, which is node block with call and return edges to each lifecycle method of the Component. This init block is in turn called and returns to a special unique node of AICCFG called, the *dispatch* node. All the asynchronous ICC calls and dispatch semantics are modeled via this dispatch node. There are two kinds of edges shown in the figure. Firstly, solid edges showing lifecycle control flows across lifecycle methods of a single component and between these methods and the *init*. Secondly, dotted edges showing control flows between *inits* and dispatch node.

As can be seen from the example AICCFG, interleavings of control flows between lifecycle methods of two different Components. The graph also shows how there is no direct method invocation edge from the ICC call (L12) to the target (L10) as was the case in the control flow graph generated by other state-of-the-art analyses. Further, the missing control flow sequence (1, 2, 3) is a possible control flow sequence in AICCFG for the application.

In next section we discuss the definition of AICCFG in detail along with an algorithm to generate an AICCFG for an application. This is followed by formal semantics of Android control flow and a discussion about the limitations of the Android application model we discuss here and soundness of AICCFG with respect to the formal semantics.

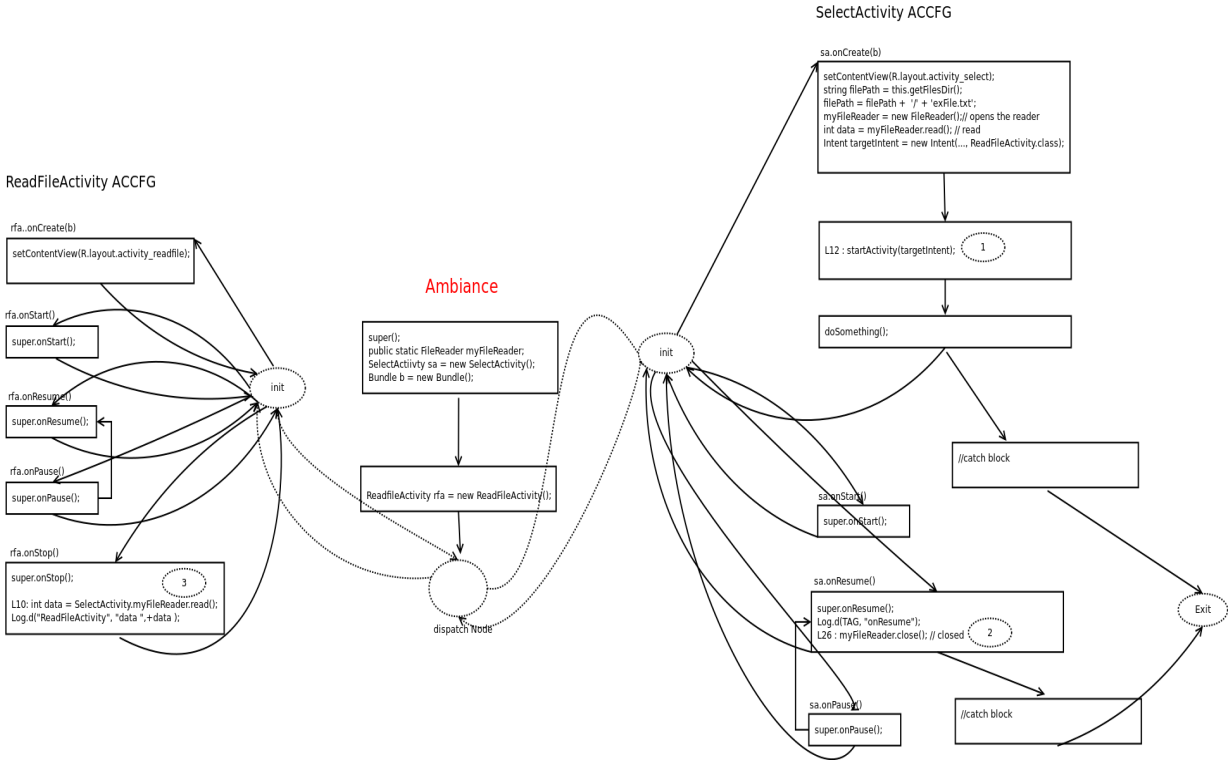


Figure 4.9: Application AICCFG showing possible flow of control

4.3 Android Application Environment Modeling And AICCFG Creation

Android applications execute inside the Android framework which asynchronously interacts with the application components on various user and system events. There exists a system dispatcher, the ActivityManagerService class of the framework which schedules an event-handler or a component lifecycle callback method based on the event fired and the state of the application. For example, in the FileReader application example, pressing of the *back* button by the user on the ReadFileActivity screen will make the Android system call ReadFileActivity's onStop method. To soundly model the control flow of Android applications we model the framework, over-approximating its asynchronous semantics and dispatch logic.

Our Android application environment model is an asynchronous control flow graph for the whole application capturing the single threaded, non-preemptive control flow semantics of these applications. This graph is called the Android Inter-Component Control Flow Graph (AICCFG). This model contains three major structures namely the *ambiance*, an *init* code block for each component instance and a set of intra- and interprocedural edges. A node of the

AICCFG models a control location or a set of control locations. The edges represent a sequence of program statements or simply a flow of control between nodes.

As explained before in Chapter 2, an Android application runs inside the Android framework and interacts with the Android system during its execution. The Android system reads the Manifest associated with an application, and controls creation, lifecycle and other control flows in the application. We discussed these features in the Android formal semantics section above. The ambiance described here models these interactions of an application with the Android system. It creates instances for components, keeps track of pending asynchronous calls, and performs other tasks performed by the Android system. Further, the ambiance models the asynchronous call and return semantics of ICC calls via a special dispatch node. The dispatch node is crucial for a sound asynchronous static analysis of an application. (refer Chapter 2). Thus, *ambiance* is the core component of an AICCFG and it models the single threaded, non-preemptive, asynchronous call semantics of an application.

Figure 4.10 shows the ambiance (middle) and lifecycle state machine of two Activities (on sides) for the example *FileReader* application. There are three kinds of edges- the dashed black edges representing the asynchronous call dispatch and return edges, solid black edges, representing inter-procedural, synchronous call and return edges, and solid grey edges representing other intraprocedural control flows.

4.3.1 Ambiance

Consider Figure 4.10. The ambiance (middle) comprises of three nodes v0, v1 and vd, and two code blocks b1 and b2, where:

- **Block b1**- Each public and launcher components in an Android application is instantiated by the Android framework when needed (e.g., launching of an application from the home screen creates an instance of the launcher component). The ambiance, contains instructions instantiating each of these components explicitly. Block b1 in the figure represent such an instantiation code block. For example, SelectActivity in the running example is a public component of the application (as declared in the *Androidmanifest.xml*) file associated with the application, thus the ambiance instantiates an object of SelectActivity class in code block b1. The following code fragment shows the code block b1 generated for the running example application.

```
1  b1{
2    SelectActivity sa1 = new SelectActivity();
3    }
```

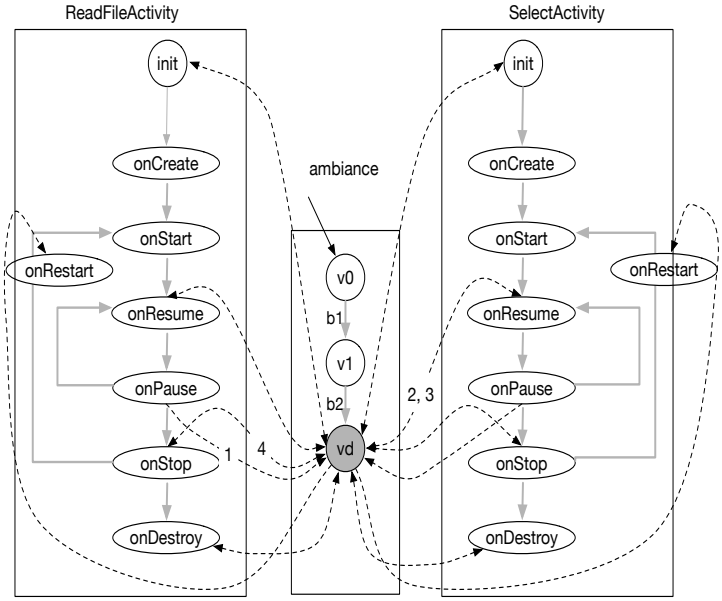


Figure 4.10: Ambiance and lifecycle state machines for FileReader Application

- **Block b2**- For each inter-component communication (ICC), explicit or implicit, the Android framework creates a new instance of the target component at runtime if this is the first ICC call to the target. Corresponding to this, the ambiance contains instructions instantiating each possible targets of all the ICCs in the application. Block b2 in the figure denotes this instantiation code block. For example, ReadFileActivity in the running example is a target of an ICC at line number 12, and the ambiance contains instructions instantiating an object of ReadFileActivity which is then invoked asynchronously from the dispatch node vd(defined next). The following code fragment shows the code block b2 generated for the running example application.

```
1  b2{
2    ReadFileActivity rfa1 = new ReadFileActivity();
3
4
5  }
```

- **Node vd**- Android framework asynchronously calls each component instance (instantiated in blocks b1 or b2) based on various user and system events, like launching of the application, pushing back button on the home screen, etc. The ambiance needs to soundly model these asynchronous calls and callbacks. Node vd in the figure is called the dispatch node and is a special control location which statically models these asynchronous calls and returns from the framework. For example, the semantics of the Android system making

an asynchronous call to ReadFileActivity's onCreate method is captured via this *dispatch* node vd. Following code fragment shows a simplified code generated for node vd. The node makes calls to initialization methods of each of the components instantiated in code blocks b1 and b2.

The lifecycle initialization method called init() in Figure 4.10 models the calls to a sequence of callback methods once the component is instantiated. For example, once an Activity is instantiated, its *onCreate, onStart, onResume* and *onPause* methods are invoked by the framework in that order, before the control can switch to another component. Edges from vd to *SelectActivity's onCreate* and other callback methods model asynchronous dispatch (and return wherever allowed by lifecycle rules) edges to various lifecycle callback methods from the framework. This is shown via a pseudo code for a dummy *main* method for the application in following code fragment. The code fragment is in *jimple* intermediate representation of Soot [119].

```
1
2
3  public static void main(java.lang.String[])
4      {
5          java.lang.String[] l0;
6          bottom_type l1;
7          com.iisc.androidanalysis.typestatetest1.SelectActivity l2;
8          com.iisc.androidanalysis.typestatetest1.ReadFileActivity l3;
9          android.os.Bundle l4;
10
11     // created during construction of b1 and b2
12         l0 := @parameter0: java.lang.String[];
13         l2 = new com.iisc.androidanalysis.typestatetest1.SelectActivity;
14         specialinvoke l2.<com.iisc.androidanalysis.typestatetest1.FirstActivity: void
       <init>()>();
15         l3 = new com.iisc.androidanalysis.typestatetest1.ReadFileActivity;
16         specialinvoke l3.<com.iisc.androidanalysis.typestatetest1.ReadFileActivity: void
       <init>()>();
17
18        // Ambiance
19       label1:
20         if l1 == 0 goto label3;
21
22         if l1 == 1 goto label3;
23
24         l4 = new android.os.Bundle;
25         specialinvoke l4.<android.os.Bundle: void <init>()>();
26         virtualinvoke l3.<com.iisc.androidanalysis.typestatetest1.ReadFileActivity: void
       onCreate(android.os.Bundle)>(l4);
27
28       label2:
29         if l1 == 4 goto label3;
```

```
30
31          if l1 == 5 goto label2;
32
33       label3:
34          if l1 == 7 goto label4;
35
36       label4:
37          if l1 == 9 goto label7;
38
39          if l1 == 10 goto label7;
40
41          l4 = new android.os.Bundle;
42          specialinvoke l4.<android.os.Bundle: void <init>()>();
43          virtualinvoke l2.<com.iisc.androidanalysis.typestatetest1.SelectActivity: void
        onCreate(android.os.Bundle)>(l4);
44
45       label5:
46          virtualinvoke l2.<com.iisc.androidanalysis.typestatetest1.SelectActivity: void
        onStart()>();
47
48       label6:
49          staticinvoke <com.iisc.androidanalysis.typestatetest1.SelectActivity: void
        <clinit>()>();
50          if l1 == 12 goto label6;
51
52          if l1 == 13 goto label6;
53
54          if l1 == 14 goto label6;
55
56          if l1 == 15 goto label7;
57
58          if l1 == 16 goto label5;
59
60       label7:
61          if l1 == 18 goto label1;
62
63          return;
64       }
65
66    }
```

Note that, these asynchronous call and return edges soundly model all possible interleavings between the callback methods of different components. Missing them may lead to unsoundness in the AICCFG. For example, Figure 4.11 shows an execution sequence which requires such a correct modeling of these inter-component flows. Such a sequence is possible when a user pushes the *back* button from the *ReadFileActivity* activity, this causes the Android system to call *ReadFileActivity's onPause*. Control can then possibly be transferred to *SelectActivity's onResume* before finally being passed to ReadFileActivity's onStop method. The execution

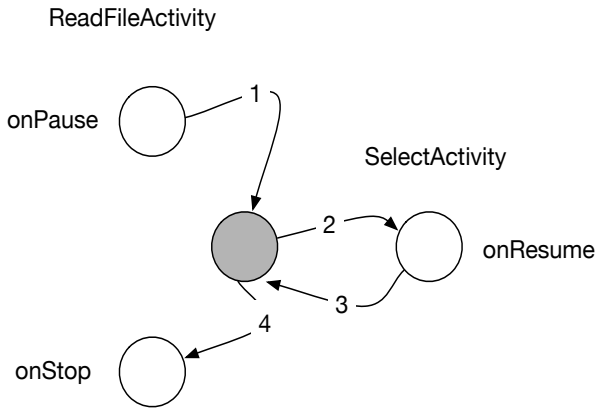


Figure 4.11: An ICC control flow interleaving for FileReader application

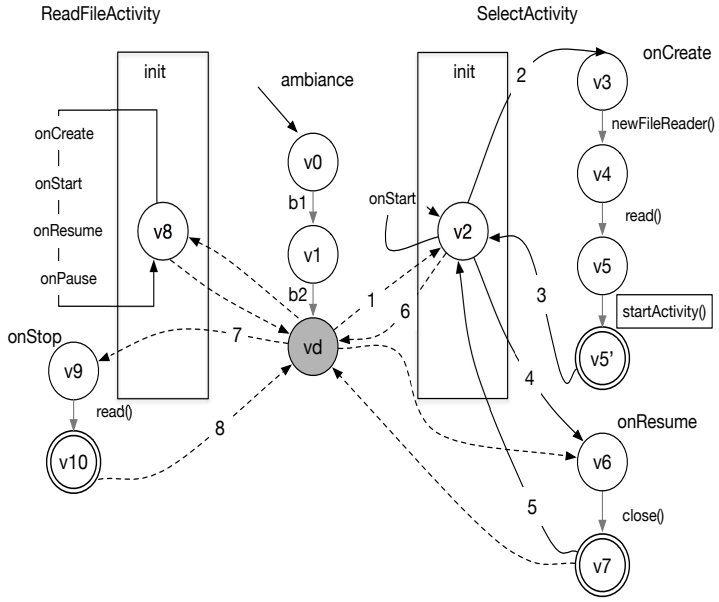


Figure 4.12: Part of AICCFG for FileReader application (⇢ : asynchronous dispatch and return edge, → : synchronous call and return edge, → : intraprocedural edges)

sequence (1-2-3-4) in Figure 4.11 can be traced in Figure 4.10 and is shown by numbered edges (1-2-3-4). Such a control flow is an outcome of interleaving of asynchronous ICC and lifecycle semantics of applications. Other works modeling lifecycle callbacks and other ICC as synchronous calls, unfortunately, miss such an interleaved execution path.

4.3.2 Android Inter Component Control Flow Graph, AICCFG

AICCFG is an asynchronous control flow graph $G_* = (V_*, E_*)$, for the whole Android application modeling the asynchronous calls, event handling and lifecycle callbacks invoked by the

Android framework. The graph is based on the asynchronous program representation described in [68] extended to model the control flows specific to Android applications. In this section we formally define the AICCFG, illustrating its features using Figure 4.12 which shows a part of the AICCFG generated for the example FileReader application. The figure is a detailed version of Figure 4.10 and shows a more fine-grained view of the control flows.

The graph in Figure 4.12 has nodes representing the control locations, with double-edged circles representing the terminal locations for a method or callback. The graph can be divided into three subgraphs for the ReadFileActivity, SelectActivity and the ambiance in the middle. There are three major types of edges in the figure-

- intraprocedural grey edges, connecting successive statements (e.g. v3 → v4).
- interprocedural solid black edges, representing synchronous call and return (e.g. v2 → v3 or v2 → v6).
- and interprocedural dashed black edges, representing asynchronous dispatch edges from the special dispatch node vd to certain lifecycle callback methods and lifecycle initialization method init() and their corresponding return edges (e.g. vd → v2 and v2 → vd).

To define the AICCFG, we begin with the definitions of substructures needed to define it. Figure 4.12 is used to illustrate these definitions.

Lifecycle Callback Control Flow Graph Each lifecycle callback method like onCreate, onStart, etc. is a Java method which is invoked by the framework on certain events and is executed atomically. The lifecycle callback control flow graph (LCCFG) defines the control flow graph for such a lifecycle callback method. It is a asynchronous control flow graph, G_{lc} = (V_{lc}, E_{lc}), where V_{lc} is the set of control locations in the callback method and E_{lc} is a set of normal control flow edges (e.g. edges like v3 → v4) along with two types of intraprocedural edges between nodes corresponding to following types of program statements-

- an intraprocedural edge from a call site to the return site corresponding to each synchronous call to a method m in the callback method. For example, edge (v4 → v5) with label *read* in the figure.
- an intraprocedural edge from a call site to the return site corresponding to each asynchronous call to a method or a component. For example, edge (v5 → v5') with a label *startActivity* in the figure.

Since an asynchronous call is stored in the task queue and not dispatched directly and the control stays with the caller, the above intraprocedural edges corresponding to asynchronous calls correctly model the delayed dispatch semantics of asynchronous calls. These asynchronous pending calls will be dispatched later from the dispatch node vd.

Android Component Control Flow Graph The Android framework enforces lifecycle rules associated with each component of an Android application. This defines the control flow semantics associated with a component. An Android component control flow graph (ACCFG), models *synchronous* part of the control flows in a component. It is a synchronous, interprocedural control flow graph $G_c = (V_c, E_c)$, which comprises of a set of LCCFG, one for each callback defined in the component along with the init method. The node set V_c is the union of the nodes V_{lc} for each LCCFG, and the edge set E_c is the union of the edges of each LCCFG, along with a set of synchronous interprocedural edges defined as follows.- For each intraprocedural edge in an LCCFG corresponding to a synchronous call, the ACCFG has -

- an interprocedural synchronous call edge from the callsite to the callee's entry node corresponding to each synchronous call in the method.
- an interprocedural synchronous return edge from the exit of the callee to the return site.

Note that the ACCFG misses the asynchronous dispatch and return edges which will be modeled by the AICCFG defined next.

Android Inter Component Control Flow Graph An AICCFG is an asynchronous control flow graph $G_* = (V_*, E_*)$, for an Android application modeling *asynchronous* calls, event handling and lifecycle callback invoked by the Android framework. Intuitively, it models the asynchronous dispatches corresponding to the asynchronous calls occurring in the application. An asynchronous dispatch is an interprocedural edge from the node vd (in the ambiance) to a method or a callback, for which there is a pending asynchronous call. For example, the edge (v5 $\rightarrow$ v5') labeled startActivity() adds a pending call to the *ReadFileActivity's init* method. This call will be dispatched later from vd, represented by the edge (vd $\rightarrow$ v8). Thus, the AICCFG integrates the ACCFGs and the ambiance together and models the correct asynchronous dispatches performed by the framework. V_* is the union of nodes for each of these elements and E_* is the union of edges for each ACCFG plus a set of the following asynchronous dispatch edges corresponding to each pending asynchronous call in the application.

- An asynchronous dispatch edge to each init $\in V_c$ of the ACCFG set. For example, the edge (vd $\rightarrow$ v2) in Figure 4.12 represents such an edge from vd to *SelectActivity's init* method.

- An asynchronous return edge from the exit of init to vd. For example, the return edge from v2 and v8 to vd in Figure 4.12
- Android framework can make asynchronous calls to certain lifecycle callback methods of a component. The AICCFG models these calls as asynchronous dispatch edges from vd to the entry of these lifecycle callback methods of each ACCFG instance. For example, edge (vd → v6) in Figure 4.12.
- Corresponding to each vd to the entry of a method like (vd → v6), there is an interprocedural return edge from the exit of callback, like (v7 → vd).

Valid Paths in AICCFG We claim, that AICCFG presents a sound over-approximation of the possible control flows in Android applications, i.e. for any possible execution sequence in an Android application, there exists a valid path (generated by concatenating nodes of AICCFG) in the AICCFG generated for the application. However, not all the paths in the AICCFG of an application is a *valid* execution sequence. A valid path in an AICCFG is defined as follows:

Definition 4.1 *A path p in an AICCFG, generated by concatenating nodes along p is a valid path iff it satisfies following properties:*

1. *p is a valid inter-procedural path, with matching synchronous call and return edges.*
2. *p satisfies all the lifecycle control flow semantics defined earlier (refer Figure 4.17) for corresponding component type. For instance, although an AICCFG might have an edge (onResume → onRestart) for some Activity component, such a control flow does not satisfy the lifecycle semantics defined for an Activity.*
3. *p has a valid asynchronous sequence ∂(p) associated with it, i.e. for each asynchronous dispatch edge in p, there must be a pending asynchronous call generating that dispatch seen earlier in p.*

Following the above definition of valid paths in an AICCFG, one can check whether a given path in an AICCFG is valid or not. For example, there is no possible execution sequence for the path (1-2-3-4-5-6-1-2-3-6), which calls the init for SelectActivity twice. This violates the third requirement in the above definition as the path does not has a valid asynchronous sequence $\partial(p)$. This is due to the fact that, each asynchronous ICC (like startActivity in edge v5 → v5') corresponds to a single asynchronous call to target. Thus informally, the set of valid paths in AICCFG, include only those paths for which every asynchronous dispatch has a matching asynchronous call defined earlier in the path. We refer the reader to [68] for a more formal semantics of asynchronous procedure calls.

4.3.3 Ambiance and AICCFG Construction

Algorithm 1 presents an algorithm for constructing the AICCFG for a given application. An Android applications is compiled into *Dalvik* executables and packaged into an *.apk* file. The application also has a *manifest* file which provides essential information about the application to the Android system. Our algorithm takes the application's apk A, and the manifest M as input and emits the AICCFG, $G_* = (V_*, E_*)$ as output. The algorithm's Main routine extracts the launchers and public components of the application from M, using auxiliary methods getLaunchers() and getPublicComps() at lines 3 and 4 respectively. The call to method getICCCalls(), at line 5 analyzes the application and the manifest and returns the set of inter-component communication calls, iCCSet in the applications. The method call to getComponentCfgs() at line 6 generates an ACCFG for each public and launcher component extracted earlier. Line 7 makes a call to subroutine createAmbiance.

The createAmbiance function (lines 11-19) takes lists of public and launcher components as input and instantiates each of these components creating blocks b1 and b2 (similar to the blocks in Figure 4.12) at lines 14 and 15. It also takes the iCCSet as input and instantiates the target of each of these ICC calls and concatenates these to b2 at line 16. It creates a new dispatch node vd at line 17 and finally concatenates each of these to the empty ambiance at line 18, and returns the generated ambiance.

The createAICCFG subroutine takes the generated ambiance and a list of ACCFGs accfgs as input and returns the final $G_* = (V_*, E_*)$ as output. It initializes the node and edge sets with empty sets (line 23) and extracts the vd from the ambiance (line 24). The outer while loop (lines 25-36) visits each ACCFG $G_c = (V_c, E_c)$, and adds the nodes and edges to V_* and E_* respectively (lines 26-28). The inner while loop (lines 30-34), looks at each node $N_c \in V_c$, and creates an asynchronous dispatch edge (vd, N_c), iff N_c is an entry node of a callback (e.g. node v3 in Figure 4.12) and adds this edge to the edge set E_* (line 33). Else, it creates an asynchronous return edge (N_c, vd), iff N_c is an exit node of a callback (e.g. nodes v7 or v10 in Figure 4.12) and then adds this edge to the edge set E_* (line 37). The method returns the G_*, to the caller Main which finally return the generated AICCFG G_*.

To understand the soundness (refer Chapter 2) of our AICCFG against the Android environment models used by other static analysis works, consider Figure 4.13 which presents a simplified partial environment model generated by IccTA [76] for the same example FileReader application. AmanDroid [125] has a similar ICC and asynchrony-unaware semantics as modeled in this graph. The figure has nodes and edges defined similarly to Figure 4.12, but lacks any asynchronous features (dispatch node and dispatch edges). Consider a path in Figure 4.12

Algorithm 1: Algorithm AICCFG construction

input : Application *Manifest* M and apk A.
output: AICCFG $G_* = (V_*, E_*)$ for the application.

```
1  Main() begin
2      launchers ← getLaunchers(M);
3      publicComps ← getPublicComps(M);
4      iCCSet ← getICCCalls(M, A);
5      accfgs ← getComponentCfgs(launchers, publicComps);
6      ambiance ← createAmbiance(launchers, publicComps, iCCSet);
7      aiccfg ← createAICCFG(ambiance, accfgs);
8      return aiccfg;
9  end
10 createAmbiance(launchers, publicComps, iCCSet) begin
11     ambiance ← ∅;
12     b1 ← instantiate(launchers);
13     b2 ← instantiate(publicComps);
14     b2 ← v1.concat(instantiate(iCCSet));
15     vd ← newNode();  /*Creates a new empty node */
16     ambiance ← ambiance.concat(b1, b2, vd);
17     return ambiance;
18 end
19 createAICCFG(ambiance, accfgs) begin
20     V* ← ∅; E* ← ∅;
21     vd ← (ambiance.vd);
22     while accfgs has Gc = (Vc, Ec) do
23         Gc ← remove(accfgs);
24         V* ← V* ∪ Vc;
25         E* ← E* ∪ Ec;
26         while Vc has Nc do
27             Nc ← remove(Vc);
28             if Nc is a callback method's entry then
29                 (vd, Nc) ← createEdge(vd, Nc);
30                 E* ← E*∪ {(vd, Nc)};
31             end
32             else if Nc is a callback method's exit then
33                 (Nc, vd) ← createEdge(Nc, vd);
34                 E* ← E*∪ {(Nc, vd)};
35             end
36         end
37     end
38     return G*;
39 end
40 getLaunchers(Manifest M) begin
41     tagValue ← parseManifest(M);
42     componentList ← getValueFromTags((action,"MAIN"),(category, "LAUNCHER");
43     return componentList;
44 end
45 getPublicComps(Manifest M) begin
46     tagValue ← parseManifest(M);
47     publicList ← getValueFromTags((exported,"true"));
48     publicListExtended ← getOuterElement((intent-filter));
49     fullPublicList ← publicList ∪ publicListExtended;
50     return fullPublicList;
51 end
```

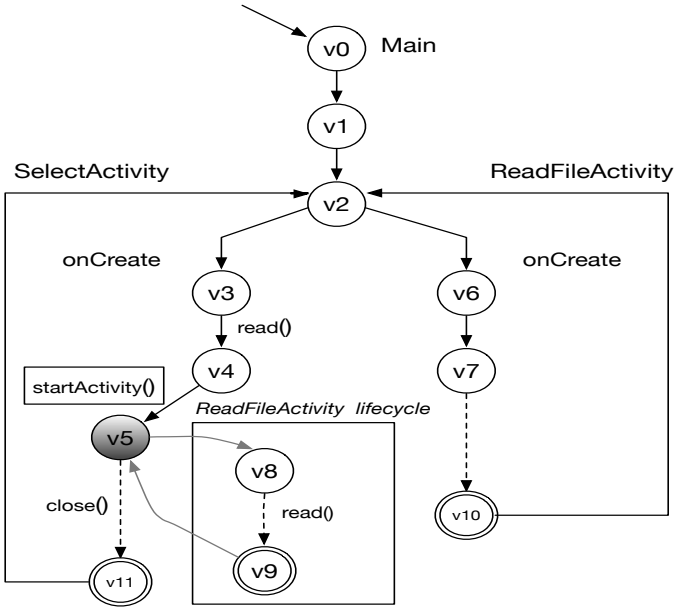


Figure 4.13: Partial Android environment model generated by IccTA for the FileReader application

with numbered edges (1-2-...-8). This is a possible valid execution path for some run of the application. The set of possible paths in Figure 4.13 lacks such a path due to the unsound modeling of asynchronous calls as synchronous. Their graph dispatches the ICC call startActivity() (edge v4 → v5) synchronously and blocks till the callee returns. Thus the call to read in ReadFileActivity's onResume() (v8 → v9) is reached before the object is closed in SelectActivity's onResume is called (v5 → v11). As evident from this example, this makes the analyses built over their model inherently unsound. In this case, the analysis will be missing a possible typestate violation at (v8 → v9).

Another source of unsoundness in IccTA comes from modeling component lifecycle as a synchronous control flow block (e.g., the ReadFileActivity's lifecycle block), which lacks the ability to model the interleaved control flows between component callbacks. For example, consider the execution sequence shown earlier in Figure 4.11. There is no possible path in Figure 4.13 to model such a sequence, while our AICCFG can easily capture such a path (4-5-6-7-8) in Figure 4.12.

4.4 Android Application Formal Modeling and Formal Control Flow Semantics

Android application developers develop applications to run in synergy with the Android framework and the Android System. Thus, the exact semantics of these applications is not straightforward and requires understanding. In this section of the chapter, we present a formal abstract syntax and a small step operational semantics for an Android application. Following this, we also present a formal semantics for Android control flow including the component lifecycle, asynchronous calls and other control flow features of an Android application. We consider an Android application as Java programs with additional control flow semantics, this is contrary to other formal modeling attempts for Android [103], which define some form of semantics over dalvik bytecode instructions. This allows us to focus away from concrete dalvik instruction, and to focus exclusively on abstract features of Android applications and their control flow.

4.4.1 Syntax

The abstract syntax for an Android application consists of a *Manifest*, a set of *Class declarations*, *Method declarations*, *fileds* and *Constructor declarations*, etc. The abstract syntax is presented in Table 4.1. We first explain the metavariables used in the syntax. A, B, C, D, E,... range over class names. f, g, h,... range over field names, m, ranges over method names, while $\sigma_m, \sigma_n, ...$, ranges over event and lifecycle handler names. x ranges over variable names, d, e range over expressions. The set of variables include a special variable this, which represent the current reference or location and cannot be used as an argument or a return variable to a method or an event-handler.

We write $\overline{\mathrm{C}}$ as a shorthand for possibly empty sequence of $\mathrm{C}_1, \mathrm{C}_2, ...\mathrm{C}_n$ and similarly define $\overline{\mathrm{f}}, \overline{\mathrm{x}}, \overline{\mathrm{M}}$, etc. The comma (,), represents a normal concatenation. We represent a pair as (a, b) and abbreviate operations on pairs of sequences in an obvious way. For instance, we use $\overline{\mathrm{Cf}}$ to represent ($\mathrm{C}_1\mathrm{f}_1$ $\mathrm{C}_2\mathrm{f}_2$ ... $\mathrm{C}_n\mathrm{f}_n$ and $\overline{\mathrm{Cf}}$; to represent ($\mathrm{C}_1\mathrm{f}_1$; $\mathrm{C}_2\mathrm{f}_2$; ...$\mathrm{C}_n\mathrm{f}_n$;). The expression, this.$\bar{\mathrm{f}}$ = $\bar{\mathrm{f}}$; is an abbreviation for, this.f_1 = f_1;...this.f_n = f_n. For simplicity, we assume no duplication of names of fields, methods, classes, variables or other names.

An android application is a list or a collection of *class declarations* $\overline{\mathrm{C}}$ and an XML property file called the *Manifest*. The *Manifest* is further defined as a list of (tag, value) pairs defining properties related to an application, its components and its environment. A *class* is defined in a syntax similar to the Feather Weight Java(FWJ) [65], extending a superclass. Contrary to the FWJ, the superclass can be of two kinds, either a normal Java class (JavaC) or a class from a set of classes called *Component Classes*. The Component classes define the base classes

(Application)	(A)	::=	(Manifest, $\overline{\text{C}}$)
(Manifest)	(Manifest)	::=	$(\overline{\text{tag, value}})$
(Class)	(C)	:=	CompC
			| JavaC
(Component Class)	(CompC)	::=	class D extends Comp { $\bar{\text{K}}$, $\bar{\text{C}}$ $\bar{\text{f}}$; $\bar{\sigma}$ $\bar{\text{M}}$ }
(Java Class)	(JavaC)	::=	class D extends E { $\bar{\text{K}}$, $\bar{\text{C}}$ $\bar{\text{f}}$;, $\bar{\text{M}}$ }
(Component)	(Comp)	::=	Activity
			| Service
			| BroadcastReceiver
			| ContentProvider
(Constructor)	(K)	::=	C ($\overline{\text{C g}}$) { super($\bar{g}$); this.$\bar{\text{f}}$ = $\bar{\text{g}}$;) }
(Method)	(M)	::=	C m ($\overline{\text{C x}}$) { $\overline{\text{C f}}$; $\bar{\text{e}}$; return e; }
(Event Handler)	(Σ)	::=	void σ_m ($\overline{\text{C f}}$) { $\overline{\text{C g}}$; $\bar{\text{e}}$; skip;}
(expression)	(e)	::=	x | value | l_i
			new K ($\bar{\text{e}}$)
			| e ; e
			| x = e
			| invoke (code, data)
			| invoke$_\bigstar$ (code, data)
			| skip
			| c
(code)	(code)	::=	M | Σ
(data)	(data)	::=	value $\setminus$ { M , Σ }
(value)	(value)	::=	c | $O^\dagger$ | M | Σ
(constant)	(c)	::=	0,1... | s $\in$ String | b $\in$ Bool |

Table 4.1: Abstract Syntax for Android Applications

for Android Components, *viz.* Activity, Service, BroadcastReceiver and ContentProvider. We represent this class set as *Comp*. The body of the class contains a list of constructors $\overline{\mathrm{K}}$, a set of fields $\bar{\mathrm{f}}$ tagged with respective classes $\bar{\mathrm{C}}$, a set of method declarations $\overline{\mathrm{M}}$ and a set of event handlers declarations $\overline{\Sigma}$, exclusively for a class extending a CompC.

An Android class has a list of method declarations $\overline{\mathrm{M}}$. Each method declaration has a return Type(Classname for the return Type), a method name m, a list of expressions as arguments $\bar{e}$ and a *body*. The method body contains a list of fields or local variable declarations, $\bar{f}$, list of expressions or statements $\bar{e}$, and a mandatory *return* statement marking the end of the method body.

An event handlers declaration Σ is similar to method declaration with one major difference, each event handler's body ends with a special "skip" expression marking the end of the event handler body. This is helpful in defining the semantics of these event handlers as an atomic block of instructions. There are some predefined events generated by Android System and analogous to them there exist predefined event handlers, these include { *onCreate, onRecieve, onStart etc*}. Apart from these Σ includes other System and user-generated event handlers.

An expression e is either a *variable* (x), a value, an abstract location represented by l_i, a constructor invocation, an assignment, an asynchronous invocation of an event handler σ, and a synchronous or an asynchronous invocation of a method m. A value could be a constant c, an abstract object $o^{\dagger}$ a method declaration m or an event handler declaration σ. A constant c could further be an integer literal, a string literal or a boolean *true* or *false*.

4.4.2 Android Semantics

4.4.3 State of the system

To discuss and argue about the operational semantics of Android applications we define the state of the system. The runtime semantics of an Android application can be reasoned using the state or the runtime configuration of the application which we define by a 5 tuple as follows.

$$\text{State, } \omega = \langle iC, \Gamma, \Pi, \mu, pc \rangle$$

where,

- iC = Set of Active Component instances.
- Γ = Method Stack $[m_0, m_1, ...m_n, \epsilon]$ with $\top(\Gamma) = m_0$.
- Π = Defined by a multiset $\hat{M}$ of pairs $\langle \sigma_i \in \Sigma, \mu_i \in \mu \rangle$ capturing non-dispatched, pending asynchronous calls.

- μ = variable $\mapsto$ Values.
- pc = List[pc_{head} :: rest], where pc_{head} is the currently executing instructions.

An Android application at any moment during its lifetime might have zero or more active instances of its components. iC keeps track of these currently active instances. This help takes decision, to what methods or event handlers a synchronous or an asynchronous method call will be resolved to. There can be more than one active instances of a single Android Component at an instance during the lifetime of an application. Γ defines the method Stack with a special procedure $\top$, defining the currently executing method of the application. Since the methods are invoked both synchronously and asynchronously, not all the *inovke* instructions push the callee on to the stack Γ. Since at any instance there are number of pending asynchronous calls which need to be dispatched later. Π is a multiset of pairs $\langle \sigma_i \in \Sigma, \mu_i \in \mu \rangle$ where, μ_i is a sub-map of μ such that the domain of $\mu_i \subseteq \mu$ and co-domain of $\mu_i \subseteq \mu$. Intuitively, μ_i is a local *map* for the variables in the context of σ_i. For example, it includes all the formal parameters of σ_i. The set of all such μ_i can be represented as the power-set 2^{μ}. We define a counter $\hat{M} : (\Sigma \times 2^{\mu} \mapsto \mathbb{N})$, returning a natural number for each pair, denoting the number of such pairs in the multiset Π.

Intuitively, Π records all pending call to a σ_i with a local *heap* value μ_i. Finally, the *state* has a list of instructions, called as the *pc*. A *pc* has a head which denotes the currently executing instruction, and a *rest* part and a terminal ϵ.

4.4.3.1 Active Components

At any time during the execution of the application, there are a set of live instances of different Components of the application. For example, an instance of an Activity is alive when the activity is *running* and the user can interact with it. We maintain a set of these alive instances, represented here as iC. iC is a subset of abstract objects $O^{\dagger}$, such that each element of iC is an object of an Component class.

4.4.3.2 Method Stack Γ

A configuration of an application is dependent on the method stack Γ which stores the active methods and currently executing method as the top element of the stack. A method stack is a stack of *method records*. A *method record* for method m, written as ψ_m is a binary tuple $< b_m, \mu_m >$, representing the (body, data) pair of the method m. b_m represents the body of the method as a list $[i_1 :: i_2 :: ... :: i_n]$ and data is a submap of μ, mapping the parameters and receiving object reference to values. This is analogous to an Activation record for the method. Thus, $\Gamma = (\psi_{m1} :: \psi_{m2} :: ... :: \epsilon)$, where ψ_{m1} is the top of the stack and currently executing method in a context μ_{m1}. The bottom of the stack is marked by an empty *method record* , ϵ.

4.4.3.3 Asynchronously Pending Calls Π

Π defines a multiset of pending asynchronous calls to either a method m $\in$ M or an event handler $\sigma \in \Sigma$. Formally, Π is a multiset $\hat{M}$ of *pending methods and event handlers frames*, λ. A *pending methods and event handlers frame* λ is a pair ($(m_i \mid \sigma_i)$, $\mu_{m_i} \mid \mu_{\sigma_i}$), of a *code* m_i (σ_i) to be executed with submap $\mu_{m_i}(\mu_{\sigma_i})$ of μ mapping the parameters for the $m_i(\sigma_i)$ to the values.

Thus, an intent i passed to a component using *startActivityForResult(i)* will be stored in Π as a mapping from the pair $((onCreate(), \mu[l_i \rightarrow o_i \ , ...]))$ to an Integer n, where l_i is the reference to the target component's object o_i. This captures the pending asynchronous call to *onCreate* event handler of the object o_i having a reference l_i.

4.4.3.4 Store μ

We define a *store* μ, mapping program variable, fields and references to values. For example, each new instance of a class C will create an abstract object value $o^{\dagger}$ which will be mapped to a reference l by μ. Apart from the global store for the application, we also define submaps μ_c such that $dom(\mu_c) \subseteq dom(\mu)$, capturing a limited context c. Such a submap is useful in defining a local calling context for a method or an event handler.

4.4.3.5 Program counter list *pc*

Finally, we define the instruction to be executed next using a *program counter, pc*. The *pc* is not just a single instruction rather a list of instructions of the form $[i_1 :: i_2 :: i_3 :: ..., i_n]$, to be executed with the currently executing one as the head of the list.

4.4.3.6 Initial State

Each Android application when started undergoes an initialization phase. During this phase, the Android System reads the application's manifest and creates an instance of the application accordingly. This initialization phase defines the initial configuration of the application. The initial configuration is defined as follows:

Initially, the set of active components is empty. Each Android application has a *Main.LAUNCHER* component listed in its manifest. This is the component which gets created and invoked when the application is first launched, either by the user or some other application. To capture this semantic, the method stack is updated with a the entry point of the *Main.LAUNCHER* (Main.LAUNCHER.onCreate() in case of an Activity component for example). This makes the *onCreate* of the Main.LAUNCHER, the currently executing method. The initial update of Π_0 needs some elucidation. The definition uses a helper function *methods* which takes a *Class* and returns the set of public methods or event handlers described in the method. For

$$\omega_0 = \langle iC_0, \Gamma_0, \Pi_0, \mu_0, pc_0 \rangle \text{ such that,}$$

$$iC_0 = \phi$$
$$\Gamma_0 = [(Main.LAUNCHER.onCreate()) :: \epsilon]$$

$$\Pi_0 = \begin{cases} \hat{M}(\sigma_i, \phi) = 1, & \text{if} \sigma_i \in methods(Main.LAUNCHER) \\ \hat{M}(\sigma_j, \phi) = 0, & \text{otherwise} \end{cases}$$

$$\mu_0 = \phi$$
$$pc_0 = [Main.LAUNCHER.onCreate.start::\epsilon]$$

Figure 4.14: Initial State of an Application, ω_0

Component classes, the function returns the set of life-cycle event handlers for the component. Π_0 is initialized based on two patterns, for all the σ_i in the *methods(Main.LAUNCHER)*, an entry (σ_i, ϕ) is added to Π, this is captured by updating the map $\hat{M}$ for this pair. Other event handlers and methods are not added to Π, which is shown as assigning a zero value to $\hat{M}(\sigma_j, \phi)$. Note that, there may be multiple LAUNCHER components and each may have multiple event handlers. This is a sound over-approximation of the actual launching semantics. Initially, the store μ_0 is empty and pc_0 is initialized with the entry expression of the *onCreate* handler of the *Main.LAUNCHER*.

4.4.3.7 Auxiliary Methods

The definition and semantics for the abstract syntax make use of certain auxiliary methods. These methods either check some property of some used element in the semantic rules or processes the manifest and return certain results useful to define the semantics. We discuss these auxiliary methods here:

Definition 4.2 (*methods(Class)*) *Abstract method methods(Class), takes a name of a class(a regular Java class or a component class) and returns a list of methods defined in the class (for regular Java class) and a list of event-handlers definitions (for a Component class).*

Definition 4.3 (*mBody(m, Class, $\overline{v_i}$)*) *mBody(m, Class, $\overline{v_i}$), takes a method name m, a class C and a list of values for formal parameters, and returns the body for m defined in C along with the local map μ_m mapping the formal parameter variables to the given list of values. If the method is not present in the given class, the method returns an empty body and empty map.*

Definition 4.4 (*target(* $invoke_X$ *(C::*σ*,* $\overline{v_i}$*))* *We define an abstract method target, to resolve to a target for invoke* $(m_i \mid \sigma_i, \overline{v_i})$ *or* $\text{invoke}_\bigstar$ $(m_i \mid \sigma_i, \overline{v_i})$*. For example, in a static method invocation. For asynchronous calls to other components, either the call is explicit in which case the target class is provided and the target is resolved to an active instance of the target component class or in case of implicit calls, the target resolution is performed via Manifest resource.*

Definition 4.5 (*instanceOf(object, className)*) *We define another abstract method instanceOf, to check if a given object is an instance of a given class className. The check is performed dynamically.*

4.4.3.8 Operational semantics for Expressions

Figures 4.14 and 4.15, presents reduction semantics rules for *expressions* defined in the Android syntax. The rules are defined over the *state* ω of the system. Each rule has the form $\omega; e; \omega'$, representing the reduction of the state ω to ω upon execution of expression e. The rules R-New-1 and R-New-2 present rules for new Instance creation for a regular Java class and an Android Component class respectively. Both, extends the *store,* μ, with a new mapping from the reference l_c to the abstract heap object o representing the newly created object. The rule R-New-2 also adds l_c to the set iC containing the active instances. Besides this, both rules, update the fields of the class with the parameter values passed to the constructor.

The R-Var is the simplest rule which returns the value for a variable x from the heap μ. The richest and important rules are the next four rules. R-invoke-1 and R-invoke-sigma-dispatch present rules for synchronous method invocation expression $invoke(C :: m, \overline{v_i})$ for the cases of invocation of a method $m \in M$ and dispatch of an earlier invoked event-handler σ. This dispatch instruction is represented by an expression $invoke(C :: \sigma, \overline{v_i})$ for $\sigma \in \Sigma$. (R-invoke-1) gets the pair of method body b_m, and a map of formal parameters of method to values μ_m using the earlier defined auxiliary method $mBody(m, C, \mu)$. The rule updates the state of the application by bushing the method body on Γ for execution, updating(joining) the heap with the map μ_m and updating the *pc.head* to the entry instruction of the invoked method b_m. The semantics of dispatch of an event handler is similar to a method invocation with two major differences. Firstly, the rule R-invoke-sigma-dispatch checks that there is a pending asynchronous call for the pair $\langle b_{sigma}, \mu_{sigma} \rangle$ by passing the pair to the map $\hat{M}$ of Π and secondly, after the execution of the expression this map is updated by decrementing the number of pending asynchronous calls to the pair.

The rule for asynchronous call instruction R-$invoke_\bigstar$. fetches the target for the given $invoke_\bigstar(C :: \sigma, \overline{v_i})$, which will be a pair of an event handler body b_σ and a variable to value

map μ_{sigma}. It then increments the pending asynchronous calls for (b_σ, μ_{sigma}) by updating the map $\hat{M}$ of Π. Finally it updates the *pc* value to the next instruction. The rule R-return presents a standard return semantics for a return instruction e in the currently executing method m with a method body b_m and formal parameters f_i. It updated the method stack Γ and the *pc* to the callee method and next instruction respectively.

Judgment form :: $\omega; e; \omega'$

$$\text{R-New-1}\frac{\begin{array}{c} K := \text{C}(\ \overline{Cg}\ )\{\ \text{super}(\bar{g}\ ;\ \text{this}.\bar{f} = \bar{g})\} \\ C :=\ \text{C extends D}\ \{\ \bar{K}\ ;\ \overline{Cf}\ ;...\} \\ K' :=\ \text{D}\ (\overline{Cg'})\{...\} \qquad \omega := \langle iC, \Gamma, \Pi, \mu, pc\rangle \end{array}}{\omega; l_c\ =\ new\ K(\bar{e})\ ; \langle iC, \Gamma, \Pi, \mu[l_c \mapsto o, \overline{f_i \mapsto e_i}], [K.entry::pc.next]\rangle}$$

$$\text{R-New-2}\frac{\begin{array}{c} K := \text{C}(\ \overline{Cg}\ )\{\ \text{super}(\bar{g}\ ;\ \text{this}.\bar{f} = \bar{g})\} \\ C :=\ \text{C extends CompC}\ \{\ \bar{K}\ ;\ \overline{Cf}\ ;...\} \\ K' :=\ \text{D}\ (\overline{Cg'})\{...\} \qquad \omega := \langle iC, \Gamma, \Pi, \mu, pc\rangle \end{array}}{\omega; l_c\ =\ new\ K(\bar{e})\ ; \langle (iC \cup l_c), \Gamma, \Pi, \mu[l_c \mapsto o, \overline{f_i \mapsto e_i}], [K.entry::pc.next]\rangle}$$

$$\text{R-Var}\frac{\mu(x) := v_x \qquad pc := [x :: pc.next :: rest]}{\langle iC, \Gamma, \Pi, \mu, pc\rangle; x; \langle iC, \Gamma, \Pi, \mu, pc' := [v_x :: pc.next :: rest]\rangle}$$

$$\text{R-invoke-1}\frac{\begin{array}{c} \langle b_m, \mu_m\rangle := mBody(m, C, \mu) \\ b_m := [s.entry :: ... :: s.exit] \\ pc := [invoke(C :: m, \overline{v_i}) :: pc.next :: rest] \end{array}}{\omega; invoke(C :: m); \omega'}$$

$$\textbf{where}\ \omega = \langle iC, \Gamma, \Pi, \mu, [invoke(C :: m) :: rest]\rangle$$

$$\omega' := \langle iC, \Gamma' := (b_m :: \Gamma), \Pi, \mu' := (\mu \sqcup \mu_m), [s.entry :: ... :: s.exit :: pc.next :: rest]\rangle$$

Figure 4.14: Reduction semantics for Android applications (i)

4.4.4 Android Lifecycle Callbacks

Android applications follow a lifecycle for itself and each of its component instances for consistent and optimum user experience and resource usage. These lifecycles are maintained by various Android System services. The lifecycle governs the control flow of Android application

$$\text{R-invoke-sigma-dispatch}\ \frac{\begin{array}{c}\langle b_{sigma}, \mu_{sigma}\rangle := mBody(\sigma, C, \mu) \\ b_{sigma} := [s.entry :: ... :: s.exit] \\ pc := [invoke(C :: m, ...) :: pc.next :: rest] \\ \hat{M}(\langle b_{sigma}, \mu_{sigma}\rangle) \geq 1\end{array}}{\begin{array}{c}\omega; invoke(C :: sigma, \overline{v_i}); \omega' \\ \textbf{where } \omega = \langle iC, \Gamma, \Pi, \mu, [invoke(C :: \sigma, \overline{v_i}) :: rest]\rangle \\ \omega' := \langle iC, \Gamma' := (b_{sigma} :: \Gamma), \Pi' := \Pi[\hat{M}((\langle b_{sigma}, \mu_{sigma}\rangle) := \hat{M}(\langle b_{sigma}, \mu_{sigma}\rangle - 1], \mu' := \\ (\mu \sqcup \mu_m), [s.entry :: ... :: s.exit :: pc.next :: rest]\rangle\end{array}}$$

$$\text{R-Return}\ \frac{\begin{array}{c}\Gamma := [b_m :: b_n :: rest] \\ m := Cm(\overline{Cf_i})\{b_m\} \\ \mu(f_i) =: v_i \\ b_n = [\_ :: invoke(b_m, \overline{v_i'}) :: s_n :: rest]]\end{array}}{\begin{array}{c}\omega; return \quad e; \omega' \\ \textbf{where } \omega = \langle iC, \Gamma, \Pi, \mu, pc\rangle \\ \omega' = \langle iC, ((b_n :: rest), \Pi, (\mu), pc = [s_n :: rest]\rangle\end{array}}$$

$$\text{R-}invoke_{\bigstar}\ \frac{\begin{array}{c}\langle b_{\sigma}, \mu_{sigma}\rangle := target(invoke_{\bigstar}(C :: \sigma, \overline{v_i})) \\ pc = [(invoke_{\bigstar}(..) :: rest)]\end{array}}{\omega; invoke_{\bigstar}(C :: \sigma, \overline{v_i}); \langle iC, \Gamma, \Pi[\hat{M}(b_{\sigma}, \mu_{sigma}) \mapsto \hat{M}(b_{\sigma}, \mu_{s}igma) + 1)], [rest :: \epsilon]\rangle}$$

Figure 4.15: Reduction semantics for Android applications (ii)

to a great degree, hence modeling of Android control flow requires a correct model of lifecycle callbacks of application components.

In this section we present a formal control flow semantics Android Component lifecycles for Activities.

Android component life cycle is modeled as a directed graph (refer Chapter 2), such that nodes of the graph are the life cycle event handlers $\sigma \in \Sigma$, like *onCreate, onPause etc.*, while an edge (σ_1, σ_2) from node σ_1 to σ_2 represent the possible control flow from σ_1 to σ_2. We can also represent such a graph using an ordering relationship $\preceq$ over event handler set Σ, thus representing (σ_1, σ_2) as $(\sigma_1 \preceq \sigma_2)$.

Figure 2.4 represents the lifecycle callback graph for Android *Activities*, note that it includes only the major life-cycle callbacks which every Android application needs to override, there are some other events handling methods which might be called in between these calls to handle certain events which we are ignoring here to make the explanation simple. These extra event handlers could be modeled and their semantics defined in a similar way. The directed graph in Figure 4.16, shows a simplified lifecycle callback graph for Android activities. The graph correctly models any possible lifecycle callbacks control flow in Android *Activity*, meaning all valid control flows between lifecycle callbacks for any Activity will have a corresponding path in the graph.

The control flow semantics for a component lifecycle callback defines a reduction rule for each tree rooted at a node in such a lifecycle graph. Figure 4.17 list these rules for lifecycle callbacks for a typical Android Activity component. Before we discuss these semantics, we present some formal definitions needed to define these control flow semantics.

4.4.4.1 Activity Stack (α)

An Android activity looses and gains focus based on the user action, for example an activity is in *focus* when it is occupying the whole screen and user can interact with it, if we start another Activity from this Activity, the current Activity goes in the background and the new Activity occupies the screen and gets *focus*. Once the user hits the back button on the new Activity the earlier Activity again comes into focus while the current Activity is killed. Android models this stack behavior of Activities via *Activity stack*. Following definitions give a formal definition of Activity stack.

Definition 4.6 (Activity Frame (ψ)) *An Activity Frame ψ is a tuple $\langle f, s, \eta, \gamma \rangle$ which is an abstraction of each activity in the stack $[\psi_1 :: \psi_2 :: ... : \epsilon]$ of Activity frames of active Activity instances. This is called as Activity Stack . An Activity stack is an analogue of the Activity stack maintained by Android ActivityManager Service, which maintains the life cycle of Android*

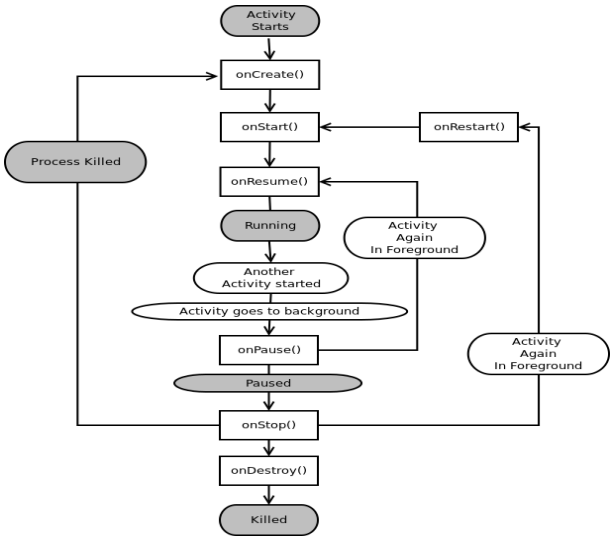


Figure 4.16: Activity lifecycle graph

Activity and other components. A in this stack defines the configuration of an activity in the activity stack, with f representing the reference to the activity, s representing the current state of the activity, where an activity state could be on of {resumed, paused, stopped or destroyed (ϕ) states}, η is the multiset of the pending activities invoked asynchronously from this activity via $invoke_{\bigstar}$ expression and finally γ is like Γ defined earlier in the context of the current activity, representing the method stack. It contains a stack of pairs ($\sigma \in \Sigma, r \in Var$) with the top of the stack representing the currently executing method and the reference to the the current activity instance.

4.4.4.2 Lifecycle reduction rules

Figure 4.17 defines the set of lifecycle transition rules for an Activity component over the configuration defined by the current Activity frame ψ. Each rule in the figure is of the form $\psi \Rightarrow \psi'$. A rule represents the transition of an Activity instance when the currently executing Activity lifecycle callback method finishes execution. For example, Rule reduce-onCreate, presents the transition of the activity frame when the Activity finishes executing the onCreate method. It says, the state of the activity goes from *stopped* to *paused* and the *onCreate* method is popped from the stack γ and *onStart* method is pushed on it, as defined by the lifecycle for an Activity. The σ_j shown in many of these rules represents possible target lifecycle method(s) based on the activity lifecycle rules of Android. For example, σ_j for reduce-onStart rule says, that the Activity lifecycle rules allow the control to flow from (onStart to onResume) or (onStart to onStop).

Next 6 reduction rules from ($reduce - onStart$ to $reduce - onDestroy$), presents similar

lifecycle control flow semantics at the end of each callback *onStart to onDesroy*. The callbacks which have more than one possible immediate successors, like *onStart and onStop*, have transitions defined corresponding to each of these successors and actual runtime trace will depend on the user or system events. The last rule *onDestroy* describes the end of the life cycle for the current activity, it updates the state of the activity as *destroyed* and the reference for the activity is destroyed and represented as $f_{\perp}$.

Judgment form :: $\psi \Rightarrow \psi'$

$$\text{reduce-onCreate}\frac{\begin{array}{c} s = stopped \\ \gamma = ((onCreate, f) :: \gamma') \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = paused : \eta : ((onStart, f) :: \gamma')}$$

$$\text{reduce-onStart}\frac{\begin{array}{c} s = paused \\ \gamma = ((onStart, f) :: \gamma') \quad \sigma_j = (onResume|onStop) \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = resumed \mid stopped : \eta : (((onResume|onStop), f) :: \gamma')}$$

$$\text{reduce-onResume}\frac{\begin{array}{c} s = resumed \\ \gamma = ((onResume, f) :: \gamma') \quad \sigma_j = (onPause) \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = paused : \eta : ((onPause, f) :: \gamma')}$$

$$\text{reduce-onStop}\frac{\begin{array}{c} s = paused \\ \gamma = ((onStop, f) :: \gamma') \quad \sigma_j = (onRestart|onDestroy) \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = stopped : \eta : (((onRestart|onDestroy), f) :: \gamma')}$$

$$\text{reduce-onPause}\frac{\begin{array}{c} s = paused \\ \gamma = ((onPause, f) :: \gamma') \quad \sigma_j = (onResume|onStop) \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = resumed| \ stopped \ : \ \eta : (((onResume \mid onStop), f) :: \gamma')}$$

$$\text{reduce-onRestart}\frac{\begin{array}{c} s = stopped \\ \gamma = ((onRestart, f) :: \gamma') \quad \sigma_j = (onStart) \end{array}}{f : s : \eta : \gamma \Rightarrow f : s' = paused : \eta : (((onStart), f) :: \gamma')}$$

$$\text{reduce-Destroy}\frac{\begin{array}{c} s = stopped \\ \gamma = ((onDestroy, f) :: \gamma') \quad \sigma_j = \phi \end{array}}{f : s : \eta : \gamma \Rightarrow f_{\perp} : s' = destroyed : \eta : (\epsilon)}$$

Figure 4.17: Control flow semantics for Activity lifecycle callbacks

Before we discuss in detail the definition and construction of a sound intermediate representation of an Android application called Android Inter-Component Control Flow Graph and discuss how it over-approximates the semantics discusses in the last section, we present a set of examples showing unorthodox control flows in an Android application and how our formal semantics model these flows.

- **Initialization**: Each Android application has an initialization phase. During this phase, the Android system reads the associated Manifest and instantiates vital components of the application. For example, the Android system finds the *Main.LAUNCHER* component of the application and creates an instance of it. Further, it makes a synchronous call to the creation callback of the component. This initialization phase is modeled by a combination of semantics rules, specifically, the initial state of the applications, parses the *Main.LAUNCHER* component and asynchronously invokes its creation callback. This increments the number of pending calls to this callback.

- **Synchronous Calls and returns**

  The basic synchronous method invocations and returns of Java semantics are captured by the invoke(code, data) expression and the semantics rules R-invoke-1 and R-Return as described in Figures 4.14 and 4.15.

- **Asynchronous ICC calls**

  One of the most important control flow features which makes the control flow in Android applications substantially different from simple Java applications and which makes other formal modeling and static analyses either unsound or imprecise is the asynchronous call semantics of Inter-Component Communication (ICC) and other asynchronous calls. The asynchrony available in these control flow features is modeled in our semantics by a dedicated asynchronous call expressions invoke$_\bigstar$(code, data) and its associated semantics discussed in operational semantics rule R-*invoke*$_\bigstar$ as described in Figures 4.15. Besides this, the asynchronous call semantics is sprinkled implicitly across different rules, like the initial state of an application, component lifecycle rules and component creation and destruction rules.

- **Lifecycle callbacks**

  As discussed earlier, each of Android application components has a lifecycle associated with it (refer Chapter 2). These lifecycles are ordered graph between various lifecycle callback methods like *onCreate, onStart*, etc. These callbacks are asynchronously called

by the Android system (Android ActivityManager Service). These callbacks, along with asynchronous ICCs, provide most of the convolution in Android application control flow making it hard to comprehend and analyze. The lifecycle associated with components is modeled explicitly in our component lifecycle semantics (refer figures 4.17). For instance, the lifecycle reduction rule for Activity's onCretae, *viz.* (reduce-onCreate) defines how the control flows from onCreate to onStart method of the Activity when the control reaches the terminal expression (skip) in onCreate.

- **Atomicity of lifecycle and other callbacks**

  Another example feature of Android applications, which affects the control flow is the atomic execution of lifecycle callbacks. Ignoring this may cause either unsoundness or imprecision. The atomicity is modeled in the semantics implicitly in each of the lifecycle callback reduction rules. For instance, the generic recursive reduction rule for Activity or Service lifecycle checks that the currently executing instruction is the *end* instruction. This instruction is tagged by the mandatory skip expression in an event handler definition.

4.5 Soundness of the Control flow Graph and Analysis

In this section, we provide the basic assumptions about the Android model we are going to discuss in this chapter, and precisely define the meaning of the term *soundness* of the Control flow graph (AICCFG) for the application and soundness of the static analysis. This will aid in a more cogent elaboration of the ideas in coming sections.

4.5.1 Assumption on Android Applications

In next two sections, we provide syntax and asynchronous control flow semantics for "single-threaded" applications with "non-preemptive" execution of lifecycle and other callbacks. This syntax and semantics captures the asynchronous calls and callbacks, Android Inter-Component communications (ICCs), Android components lifecycle and interaction between them. This assumption of single-threaded applications restricts our modeling and analysis to a subset of Android applications, yet it covers a large number of useful Android applications which are not multi-threaded. The general case of statically analyzing asynchrony and concurrency together is undecidable even for the simplest problem of reachability over control flow graph [109] and needs a different approach to analysis. We leave approximate solutions to this general problem as a possible future work.

4.5.2 Android Application Control Flow Graph

In Section 4.3, we present an Android Inter-Component Control Flow Graph (AICCFG), an intermediate program representation capturing the asynchronous control flow in Android applications. Detailed definition of AICCFG, its construction algorithm and other features are discussed in Chapter 4. Intuitively, the AICCFG captures Android asynchronous control flow, Android ICC, the Android system enforced Component and application lifecycle, event handling callbacks and other control flow governing features. The AICCFG acts as an input program to the asynchronous static analysis algorithm.

4.5.3 Soundness of Control Flow Graph

A sound static analysis for Android applications inherently requires the program representation to correctly capture Android asynchronous control flow and other features. This gives rise to the definition of "soundness" for the AICCFG.

Definition 4.7 (Soundness of AICCFG) *For a given single-threaded Android application A, with non-preemptive execution of lifecycle and other callbacks (based on the assumed model defined above), we say that the AICCFG G_A generated for this application is sound, if for any possible execution trace tr of A (a sequence of instructions), there exists a possible path p_tr in G_A, such that the string generated by the edges of the path p_tr (a sequence of instructions) is equal to tr.*

4.5.4 Soundness of a Static Analysis

"Soundness" of a static analysis is a standard term and we use it in a similar sense. A Static analysis, checking a property ϕ over a program P is sound if whenever P can violate ϕ during some execution of P, the analysis must track this violation. This definition is defined in terms of static analysis checking a property violation, a more general definition of *soundness* of static analysis can be found in [97]

4.6 Typestate Analysis

Typestate [115] is a refinement over *type*. Whereas the type of a data object defines the set of operations that are ever permitted on the object, *typestate* defines the subset of these operations that are valid in a given context. For example, Java Collections class allows getting the next element from the collection (call to Collections.next()) only if the collection has another element in the collection, else it throws an *IllegalStateException*. Static typestate analysis could be highly useful in catching programs which might be syntactically legal but meaningless or

semantically invalid[52]. In the absence of typestate analysis, the programmer needs to perform runtime checks and adhere to the API usage rules which hamper performance and is error-prone.

Android framework provides a large set of resources like Camera, MediaPlayer, Databases, etc., to be used by applications through APIs. Some of these resource APIs, (e.g., Android MediaPlayer) have a fairly complex protocol [6], making it difficult and error-prone to be enforced by the programmer. The violations of these protocols could have effects ranging from benign application crash to providing attack surfaces to attackers breaching application and user security.

Apart from the resource APIs, many other important safety properties in Android applications (like granting and revoking of UriPermissions) could be modeled and verified using a typestate analysis. Control flow soundness and precision requirements of typestate analysis make it challenging for Android applications.

4.6.1 Android Typestate Analysis

This section defines a typestate analysis for Android over the control flow semantics and the AICCFG defined earlier. The analysis is the first typestate analysis for Android applications. We model our typestate problem as an asynchronous interprocedural finite distributive subset (AIFDS) problem [68]. Although our asynchronous analysis derives from that work in theory, the following are the challenges which are specific to asynchronous inter-component analysis on Android applications-

- Android ICC calls have a complex runtime semantics and lack explicit asynchronous calls and returns. The asynchronous calls are either due to ICC or callbacks from the framework to the application. Moreover, these are asynchronous calls to a collection of callback methods rather than a single target method, which needs to be resolved either explicitly or using the manifest. Once the target is resolved, all the valid paths should be invoked based on the called component type and its lifecycle.

- Contrary to the definition of valid paths in [68], *valid* paths for the AICCFG discussed in section 4.3, are a function of both pending asynchronous calls, and application's component lifecycle rules. Thus we need a modified version of the original asynchronous data flow analysis (ADFA) algorithm to soundly calculate the meet-over-valid-path (MVP) solution over these valid paths of the application.

Our analysis like original AIFDS contains a dispatch node (vd). Since both the works try to model the *event-loop* or the *environment* using this node, there are numerous similarities

between the two. However, there are many important differences in the dispatch node semantics (or challenges unique to AIFDS for AICCFG) other than the syntactic differences listed above-

- The semantics of the event-loop (and hence the dispatch node) in Android is substantially more complex than the semantics of the general asynchronous call and dispatch as discussed in the AIFDS work. For example, consider the semantics of an Activity's *onStart* method. The onStart method can be dispatched multiple times in a life-time of an Activity based on the user interaction and life-cycle automaton associated with Activities. Thus, the dispatch node, requires to keep the *onStart* method in its pending list at all time (even after it is dispatch) before the Activity is destroyed. This is different than the one-to-one mapping with single incrementing (decrementing) of pending calls to a method on asynchronous call (asynchronous dispatch) in AIFDS. The dispatch node as modeled in the original AIFDS work does not model this.

- The above difference is not specific to a single Component type or a method, rather more generally the dispatch node in our work models the semantics associated state changes due to the lifecycle machine of the component. For example, the semantics of a return from an Activity's *onPause* method, will add a pending asynchronous calls to all the possible targets in the lifecycle machine of an Activity, i.e. {*onResume, onStop*}.

- Each component has a *destructor* method, for example *onDesroy* for an Activity. The semantics of a destructor is very different than the other methods in the lifecycle. The event-loop before destroying the component, dispatches all of the pending calls to the component's methods (dispatching multiple times if required). Thus again a dispatch to a single method may affect the semantics of the dispatch node for all other calls. The original AIFDS always has injective relation between call and dispatch of asynchronous methods.

Formally an AIFDS formulation of the Android typestate analysis problem is defined as follows-

Definition 4.8 (Typestate Property, FA) *A typestate property to be verified, is represented as a finite automaton* $FA = \langle \Sigma, Q, \delta, S, Q \setminus \{err\} \rangle$ *where* Σ *is the set of operations possible on objects,* Q *is the set of typestates a resource object might exist in,* $\delta : Q \times \Sigma \rightarrow Q$ *. err is a single error state,* S *is a unique start state and* $Q \setminus \{err\}$ *(all the states other than the error state) contains final states.*

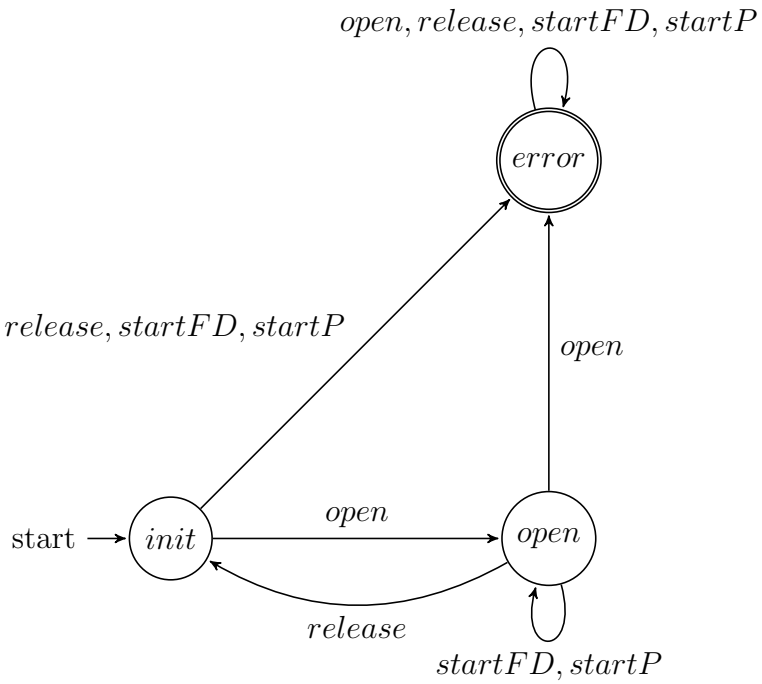


Figure 4.18: Simplified Typestate property finite automaton for Android Camera API: startFD := startFaceDetection, startP := startPreview

For example, Figure 4.18 presents a simplified version of the typestate property finite automaton for the Android *Camera* resource API. The actual Camera resource have a much larger set of public methods which we have ignored here to make the figure easier to understand.

Resource usage protocol for Android Camera resource is simpler than many other Android resources. For instance consider Figure 4.19. It shows the protocol or the typestate property finite automaton for *MediaPlayer* API in Android. The figure is adapted from the official Android portal.

Definition 4.9 (Typestate Mapping Function, α) *We define $\alpha : Ref \mapsto 2^Q$, a typestate mapping function from the object reference set Ref, to the powerset of Q. $\alpha(r_i)$ represents the set of possible typestates a given reference $r_i \in Ref$ could have at a given program point. The references set Ref is a set of symbolic object references, each of which may-point to a symbolic object instantiated at some location(intermediate representation line number) in the program. We run a standard intra-procedural may-points to analysis and must-points to analysis to associate a may-alias and must-alias set for each symbolic reference.*

Table 4.2: Typestate Transfer Functions

Transfer Functions F			
S.No.	Statement, st	$(d_g, d_l,$ CMap$) \rightarrow$ Out	Side Conditions
1	new C()	$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l \cup \{(so_i, \{FA.start\})\},$ CMap)	(C $\in$ resourceclasses $\wedge$ Statement st defines a method local reference. $\wedge$ $so_i \notin SO$)
		$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g \cup \{(so_i, \{FA.start\})\}, d_l,$ CMap)	(C $\in$ resourceclasses $\wedge$ Statement st defines an application level referenc. $\wedge$ $so_i \notin SO$)
		$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap)	otherwise
2	startActivity(target) | startXYZ(target)	$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap[$(r_t,$ init(), $d_l) \leftarrow$ (CMap(r_t, init(), d_l)++), (r_t, onX(), $d_l) \leftarrow$ (CMap(r_t, onX(), d_l)++)]	$r_t \leftarrow$ getTartget(st)
3	dispatch(m(), d_l)	$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap[$(r_t,$ init(), $d_l) \leftarrow$ (CMap(r_t, init(), d_l)−−), (r_t, onCreate(), $d_l) \leftarrow$ (CMap(r_t, onCreate(), d_l)−−)]	(m = r_t.init() $\wedge$ CMap(r_t, init(), d_l) > 0)
		$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap)	(m = r_t.onX() $\wedge$ CMap(r_t, onX(), d_l) > 0 $\wedge$ onX() $\neq$ onDestroy())
		$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap[$(r_t,$ onX(), $d_l) \leftarrow$ (CMap(r_t, onX(), d_l)−−)]	(m = r_t.onDestroy() $\wedge$ CMap(r_t, onDestroy(), d_l) > 0)
4	Normal statement	$(d_g, d_l,$ CMap$)) \rightarrow$ $d'_g, d'_l,$ CMap)	$d'_g \leftarrow$ transferTS(st, d_g, FA) $d'_l \leftarrow$ transferTS(st, d_l, FA)
5	startService(target) | bindService(target)	$(d_g, d_l,$ CMap$) \rightarrow$ $(d_g, d_l,$ CMap[$(r_t,$ init(), $d_l) \leftarrow$ (CMap(r_t, init(), d_l)++), (r_t, onX(), $d_l) \leftarrow$ (CMap(r_t, onX(), d_l)++)]	$r_t \leftarrow$ getTartget(st)

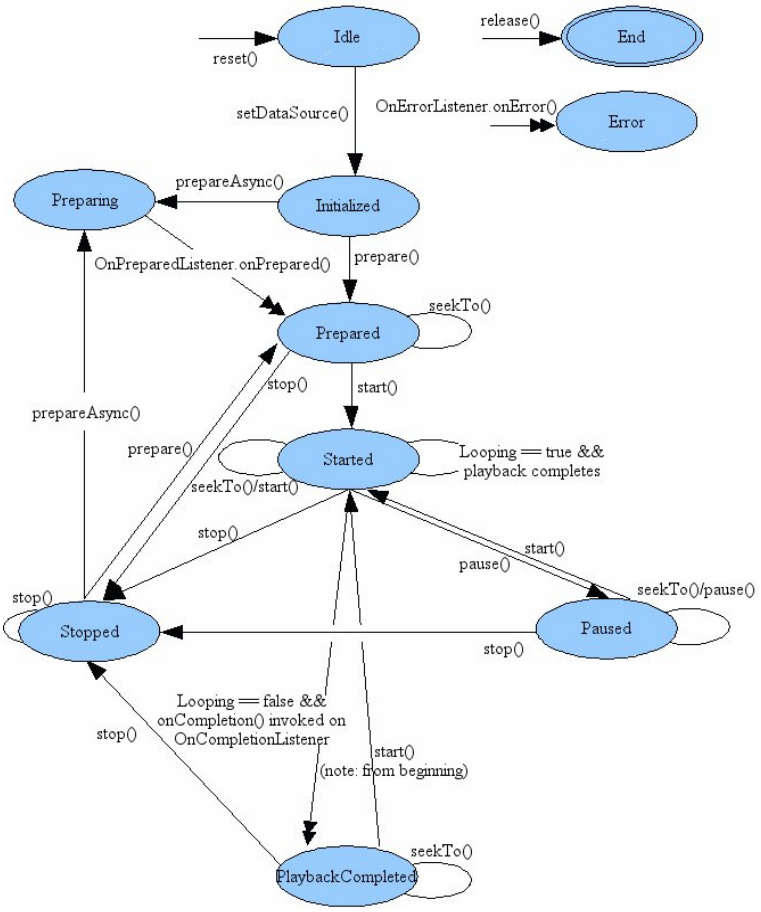


Figure 4.19: Typestate property finite automaton for Android MediaPlayer API source: [7]

4.6.2 Typestate as an AIFDS Problem

We describe the Android typestate verification problem as an AIFDS [68] instance. An AIFDS problem is an asynchronous version of an interprocedural finite distributive subset (IFDS) problem [110] for calculating MVP solutions over asynchronous programs. An AIFDS instance is a six-tuple, A = $(G^*, D_g, D_l, CMap, F, \sqcup)$, which we solve using a modified ADFA algorithm to track typestate violations for a given application and a typestate property. We define each of these tuples of A now in detail.

4.6.2.1 The Program Representation, G^*

The AICCFG G_*=(V_*, E_*), defined in section 4.3, acts as the input for our AIFDS analysis instance.

The typestate analysis also requires a typestate property automaton *FA* for the resource object as an input. In our current prototype implementation, the user needs to provide this

property automaton for the resource following the resource APIs documentation. We have provided such automata for important resource types for Android like Camera, MediaPlayer, File, SqliteDatabase, etc. in our implementation.

4.6.2.2 Data Flow Facts

Since AICCFG is an asynchronous program representation, we need to split each typestate data flow fact into global and local component rather than having a single global data flow fact. Such a division is necessary because, at the point of the asynchronous call, we need to capture incoming data flow facts, passed to the called procedure. We then store this asynchronous call as a pending call with these facts to be dispatched later. We cannot use a single global set of facts to represent the input fact for the pending call because operations that get executed between the asynchronous call and actual dispatch may change the global fact, leaving the local fact unchanged at the callsite.

The data flow facts D = $(D_l \times D_g \times CMap)$, where:

- D_g and D_l are global and local typestate dataflow facts. The data flow facts associated with *globally defined variables, static fields and class references* are treated as global dataflow facts. These are treated differently than the local data flow facts. Local data flow facts are the data flow facts associated with method local variables, fields and references. For example, a data flow fact showing a possible set of states of a global *Camera* reference visible to all components will be a global data flow fact D_g, while another such fact for a method or component local *Camera* reference, will be a local data flow fact D_l.

  Each data flow fact $d_g \in D_g$ or $d_l \in D_l$ is a pair of the form $(so_i, \{s_i, s_j ..., s_m\})$ where $so_i \in SO$. The $SO \subset Ref$ and is the set of symbolic object references for the resource objects, while Ref is the set of object references in the application. Each $s_i \in Q$, represents the possible set of typestates a given resource object so_i could be in, at a given program point.

- $CMap : (Ref \times Methods \times D_l) \mapsto N$, where Ref is the set of object references as described earlier, $Methods$ is the set of methods (callbacks, event handlers and normal methods), and N is the set of natural numbers. Intuitively the map $CMap$, captures the number of pending asynchronous calls not yet dispatched for a given method m $\in Methods$, of an object reference $r_i \in Ref$ with a given local data flow fact d_l. If there are no pending calls for the triple consisting of a given reference, method and local data flow facts, CMap maps the triple to 0.

4.6.2.3 Transfer Functions F for Typestate Analysis

Table 4.6.2.3 defines the set of transfer functions F for the problem. The columns give the rule number, the statement type, the transfer function and the side condition in which this function is applied. Each row for a given statement type is subdivided into subrows, defining the functions for a given side condition. Rule 1 applies to an object instantiation statement and Android resource initialization API calls. The rule first checks if the class being instantiated is a resource class (e.g. Android's Camera class), then it creates a new symbolic object so_i for the resource, initializes its typestate to the FA's start state S and updates the local or the global data flow facts depending upon whether the statement defines a method local reference or an application level global reference. Rule 4 defines the transfer function which applies the FA's typestate transition δ based on the incoming typestates and the operation performed on the symbolic object. The auxiliary function transferTS(), takes a statement (an operation), a data flow fact d_g/d_l (containing the current state) and the typestate property automata FA and returns an output data flow fact d'_g/d'_l (containing the target state). These operations do not alter the pending calls and hence the output CMap is the same as the incoming CMap.

Rule 2 defines the transfer functions for asynchronous calls like *startActivity, startService* or *startXYZ* in general representing an ICC call. Android framework invokes a sequence of calls (modeled by *init* method in our AICCFG) on component creation. To model this semantics, this rule increments the pending calls for (r_t, init(), d_l). Once a component is created, the framework may invoke any lifecycle callback of the component. Hence we also increment the pending calls for other callback methods (r_t, *onX()*, d_l). We have used 'onX()' for a generic callback method name (e.g. *onStart(), onResume()*, etc.). Rule 3 defines the transfer functions for the asynchronous dispatch of method m with data flow fact d_l from the special dispatch node vd of AICCFG. It checks if the dispatched method m is an initialization method *init* and has a pending call with the given d_l, then it dispatches the call and decrements the corresponding CMap cells for (r_t, *init*, d_l) and (r_t, *onCreate()*, d_l). The onCreate() is called only once during the lifetime of a component instance, hence we decrement its counter here during initialization. If the call is neither to *init* nor to a component destruction method like *onDestroy()*, the function just dispatches the call without modifying the CMap. Android Activity component callback methods other than *onCreate()* and *onDestroy()* (similarly, for creation and destruction callback methods of other component types) can be invoked any number of times during the life time of the component due to some user or system event. Hence we do not decrement the pending calls to these methods here. Finally, the call to component destruction method like onDestroy(), decrements the pending calls for all the asynchronously called callbacks and

the init() method for the given object in the CMap as the object is now dead and none of its callbacks could be invoked by the framework.

4.6.2.4 The meet operation $\sqcup$

The meet operation defines the meet of data flow facts along two or more different paths in the AICCFG. Our typestate analysis is conservative in nature and thus performs a weak update to typestate associated with any abstract object so_i. At junction nodes with two or more different incoming paths we perform a set union operation over the $\alpha(so_i)$ for each $so_i \in SO$. Formally, for any merge node p in the AICCFG of the application and for each abstract object so_i, the meet $\cup$ is defined as a union over the $\alpha(so_i)$ over all the valid interprocedural paths p_i starting from the start of the ambiance and merging at p.

For our analysis, we modified the ADFA algorithm(refer Chapter 2) [68], to handle asynchronous calls and dispatch semantics of Android applications. The major modifications are needed to handle the different semantics of asynchronous calls and dispatches discussed in the original ADFA algorithm compared to the semantics of Android applications calls and dispatch. For instance, while the original algorithm assumes explicit asynchronous calls and return points, ICC's and other asynchronous calls in Android application are not explicit, further an asynchronously called or dispatched method (or a single call can cause multiple invocations) may lead to further set of chained asynchronous calls. Besides, an asynchronously called/dispatched method(s) do not have an explicit return points.

Second major difference arises due to the lifecycle semantics of Android components described earlier. This effects the way Counter maps are updated by the algorithm. For example, unlike the original ADFA algorithm, where an asynchronous dispatch will decrement the number of pending asynchronous calls for a pair of method and local data flow facts, in Android, a dispatch may not effect the number of pending calls as the lifecycle allows further execution of an lifecycle handling method. This is a common scenario in case of an Activity component, where the lifecycle is intricate.

We then ran the modified ADFA to calculate $MVP(I_1)$ and $MVP({I_1}^{\infty})$, refer [68]. We did not need to run for higher values of k=2,3... as the $MVP(I_1)$ and $MVP({I_1}^{\infty})$ converge for our typestate analysis problem over the benchmark applications we ran on.

4.7 Implementation

The overall implementation of our approach has two major components- (1) AICCFG generator and (2) A typestate analysis for Android applications.

4.7.1 AICCFG Generator

The AICCFG generator generates a sound AICCFG for the application using the AICCFG algorithm 1. The algorithm uses certain auxiliary methods like *getLaunchers, getPublicComps, getICCCalls, and getComponentCfgs*. These methods gather information and create graphs for each component which acts as input to the *createAmbiance* and *createAICCFG* routines. We explain, each of these briefly:

- **getLaunchers** : Android applications have one or more components defined as *LAUNCHER* or *MAIN* components of an application. These components act as main entry points to an application when the application is invoked by a user. A component is registered as a LAUNCHER/MAIN component in application's Manifest, using *intent-filters* for *action* and *category* tags. For example,

  ⟨ action android:name="android.intent.action.MAIN" /⟩

  ⟨ category android:name="android.intent.category.LAUNCHER" /⟩

  The *getLaunchers* method, parses the Manifest file and finds an activity or other components registered as MAIN or LAUNCHER components using above intent-filters, and returns a list of such components. The *createAmbiance* routine uses this information to generate instructions which instantiate these components.

- **getPublicComps**: Not all the components of an Android application are visible to the outer world for invocation. A component can be invoked by a user or another component, via an ICC, only iff it is a *public* component. A component shows its intention of being public by registering for receiving external intents (refer Chapter 2). This is done either via setting the value of *exported* tag as TRUE, or by defining an *intent-filter* for the component. A component registered for some intent, using an intent-filter is implicitly treated as a public component by the Android system. Such components may act as other entry points to the application on some ICC from other applications or Android system processes. The *getPublicComps* method, parses the Manifest file and finds such public components by looking for appropriate tags and returns a list of such components. The *createAmbiance* routine uses this information to generate instructions which instantiate these components.

- **getICCCalls**: The *ambiance* need to create instructions to instantiate each component which might be invoked via some asynchronous ICC, from a launcher or a public compo-

nent. This helps the *createAICCFG* method to generate a complete control flow graph for the application. The *getICCCalls* method performs a static analysis of the application to find such components. It performs an intraprocedural string analysis and a flow-insensitive points-to analysis, to search for component names that are passed as arguments to some ICC API calls like, *startActivity, startService, startActivityForResult, etc.* (in case of explicit ICC calls), and for action and data tags of such ICC calls (in case of implicit ICC calls). The *createAmbiance* routine uses this information to generate instructions which instantiate these components.

- **getComponentCfgs**: The *getComponentCfgs*, generates a component control flow graph for each component in the application. The method takes as input the earlier generated list of public and launcher components and generates instructions for control flows, which model life-cycle associated control flow for the component. It then uses known static control flow graph generation techniques (for example on-the-fly cfg generation from Soot [119]) to generate a precise control flow graph for each component. The *createAICCFG* method, uses these cfgs to generate Android inter-component control flow graph (AICCFG).

4.7.2 Typestate Analysis

The typestate analysis is modeled as an instance of AIFDS problem [68] and is solved using a modified ADFA [68] built using Soot's *heros* IFDS implementation [119]. The typestate analysis, uses the AICCFG generated by the AICCFG generation algorithm as the program input and implements the transfer functions presented in table I. The implementation solves the AIFDS instance of the typestate problem using a variant of the ADFA algorithm for Android applications calculating under- and over-approximate MVP results.

4.8 Evaluation

4.8.1 AsyncBench Benchmarks

We present a set of synthetic benchmark applications, AsyncBench [4], containing tests for typestate violations whose verification requires a sound modeling and tracking of control and data dependencies in Android applications including the asynchronous semantics and sound lifecycle modeling. Although the benchmarks are small applications, the sound asynchronous control flow modeling presented by us is generic and we believe that the AICCFG construction algorithm is scalable. We leave the scalability studies and further evaluations of our client typestate analysis as future work. Table 4.3 concisely presents the salient features of each type of

application for each resource category. The four types of applications in each resource category test the correct modeling of asynchronous calls and lifecycle interleavings as discussed earlier in Section 4.3, and the precision of these models (via benign applications with no typestate violations).

App-type	Salient Features
Type0	No typestate violation.
Type1	Single typestate violation, **requires** sound asynchronous semantics modeling.
Type2 and Type3	Single typestate violation, **requires** sound asynchronous semantics modeling and sound lifecycle interleaving across components. These two types model two different interleavings between callbacks.

Table 4.3: Benchmark Applications

The aim of these benchmarks is two-fold. First, they check the coverage and soundness of the analysis for asynchronous calls and lifecycle properties of applications. Second, we add applications in various categories using different Android resources, namely *Camera, SQLiteDatabase, MediaPlayer, Files, etc.* This checks the usability angle of our analysis and shows that the analysis is generic enough to capture typestate violations against a rich set of Android resource usage protocols. Towards the similar goal, we also have applications, modeling the safe granting and revocation of Android UriPermissions. Android framework provides UriPermissions, which allows an application to grant temporary read/write access permissions (for its resource) to other applications. These UriPermissions are useful for ContentProviders to grant permissions to other applications to temporarily access some or whole of their data. This is done either by setting an Intent flag like, Intent.FLAG_GRANT_READ_URI_PERMISSION or by invoking the grantUriPermission() method of the Android ContextWrapper class. One possible bug in the usage of these temporary UriPermissions is the leak of these permissions when the grantor forgets to revoke the permissions through corresponding revokeUriPermission() call. Typestate can easily model and check such permission leaks by checking any UriPermission granted temporarily is always revoked before the granting component is terminated and a URIPermission is not revoked without being granted before. Next, we discuss one of these benchmarks briefly, we have made these benchmarks publicly available online [4] with comments explaining the details of typestate properties associated or being violated.

4.8.1.1 Camera Applications

This set of applications use Android *Camera* APIs. The typestate property associated with the Android Camera resource can be modeled (in a simplified way) via the typestate property finite automaton presented in Figure 4.18. The applications fall into four application types as discussed in Table 4.3 and require the analysis to soundly capture the asynchronous ICC semantics, lifecycles semantics and possible interleaving between them. The Cameraapp_0 application has no typestate violation while the other three have a single typestate violation. Listing 4.1 shows the code fragment for the Cameraapp_1. The application consists of two activities, *viz.*, FirstActivity(FA) and CaptureImageActivity(CIA). The application contains a typestate violation in the CaptureImageActivity's onStart method (line number 30, 31) caused due to control flow from *FA's onCreate, FA's onResume* to *CIA's onStart* method. The state-of-the-art, asynchrony-unaware static analysis works handle the ICC call at line number 50 synchronously and thus miss the possible state change of the Camera resource occurring at line 60 in FA's onResume method occurring before the actual dispatch of the ICC. We compare our approach against other asynchrony-unaware static analysis works in next section. Other Camera benchmark applications and applications in other resource categories are publicly available [4] with required comments explaining the details of the typestate property being verified and possible violation of the property.

Listing 4.1: Code Fragment for Cameraapp_1 Application from AsyncBench Suite

```
1  package com.iisc.android.typestatebenchcamera_01;
2  import android.support.v7.app.ActionBarActivity;
3  import android.support.v7.app.ActionBar;
4  import android.support.v4.app.Fragment;
5  import android.content.Context;
6  import android.content.Intent;
7  import android.content.pm.PackageManager;
8  import android.os.Bundle;
9  import android.util.Log;
10 import android.view.LayoutInflater;
11 import android.view.Menu;
12 import android.view.MenuItem;
13 import android.view.View;
14 import android.view.ViewGroup;
15 import android.os.Build;
16 /*
17  * @description - App violates the typestate property of Camera,
18  * @requires - Sound model of Asynchronous call
19  * @unsoundness_leadsto - FN - (FA.onStart -> FA.onResume -> CIA.onStart )
20  */
21 public class FirstActivity extends ActionBarActivity {
22       public static final String TAG = "FirstActivity";
23       public static android.hardware.Camera mycam= null;
```

```
24      @Override
25      protected void onCreate(Bundle savedInstanceState) {
26         super.onCreate(savedInstanceState);
27         setContentView(R.layout.activity_first);
28         Log.i(TAG, "onCreate");
29
30      }
31      @Override
32      protected void onStart() {
33         // TODO Auto-generated method stub
34         super.onStart();
35         Log.i(TAG, "onStart");
36         if(checkCameraHardware(this))
37            Log.d(TAG, "The device has a camera");
38         else
39            Log.d(TAG, "No Camera on the device");
40          try {
41            mycam = android.hardware.Camera.open(0); // attempt to get a Camera instance
42            Log.d(TAG, "Camra "+mycam);
43          }
44          catch (Exception e){
45              // Camera is not available (in use or does not exist)
46            Log.d(TAG, "Exception"+ e.getMessage());
47          }
48            Log.d(TAG, "Cam "+mycam);
49         Intent startCaptureActivity = new Intent(this, CaptureImageActivity.class);
50         startActivity(startCaptureActivity);
51         mycam.startPreview();
52
53      }
54
55
56   @Override
57   protected void onResume() {
58   // TODO Auto-generated method stub
59      super.onResume();
60      mycam.release();
61   }
62
63   @Override
64   public boolean onCreateOptionsMenu(Menu menu) {
65
66      // Inflate the menu; this adds items to the action bar if it is present.
67      getMenuInflater().inflate(R.menu.first, menu);
68      return true;
69   }
70
71   @Override
72   public boolean onOptionsItemSelected(MenuItem item) {
73      // Handle action bar item clicks here. The action bar will
74      // automatically handle clicks on the Home/Up button, so long
75      // as you specify a parent activity in AndroidManifest.xml.
76      int id = item.getItemId();
```

```
77        if (id == R.id.action_settings) {
78           return true;
79        }
80        return super.onOptionsItemSelected(item);
81     }
82
83     /**
84      * A placeholder fragment containing a simple view.
85      */
86     public static class PlaceholderFragment extends Fragment {
87
88        public PlaceholderFragment() {
89        }
90
91        @Override
92        public View onCreateView(LayoutInflater inflater, ViewGroup container,
93              Bundle savedInstanceState) {
94           View rootView = inflater.inflate(R.layout.fragment_first,
95                 container, false);
96           return rootView;
97        }
98     }
99
100    private boolean checkCameraHardware(Context context) {
101        if (context.getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA)){
102            // this device has a camera
103            return true;
104        } else {
105            // no camera on this device
106            return false;
107        }
108    }
109
110 }
111
112
113 %
```

```
1  package com.iisc.android.typestatebenchcamera_01;
2
3  import android.support.v7.app.ActionBarActivity;
4  import android.support.v7.app.ActionBar;
5  import android.support.v4.app.Fragment;
6  import android.os.Bundle;
7  import android.view.LayoutInflater;
8  import android.view.Menu;
9  import android.view.MenuItem;
10 import android.view.View;
11 import android.view.ViewGroup;
12 import android.os.Build;
13
14 public class CaptureImageActivity extends ActionBarActivity {
```

```
15
16      @Override
17      protected void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         setContentView(R.layout.activity_capture_image);
20
21         if (savedInstanceState == null) {
22            getSupportFragmentManager().beginTransaction()
23                  .add(R.id.container, new PlaceholderFragment()).commit();
24         }
25      }
26
27      @Override
28      protected void onStart() {
29         // TODO Auto-generated method stub
30         super.onStart();
31         FirstActivity.mycam.startPreview();
32         FirstActivity.mycam.startFaceDetection();
33
34
35      }
36
37      @Override
38      public boolean onCreateOptionsMenu(Menu menu) {
39
40         // Inflate the menu; this adds items to the action bar if it is present.
41         getMenuInflater().inflate(R.menu.capture_image, menu);
42         return true;
43      }
44
45      @Override
46      public boolean onOptionsItemSelected(MenuItem item) {
47         // Handle action bar item clicks here. The action bar will
48         // automatically handle clicks on the Home/Up button, so long
49         // as you specify a parent activity in AndroidManifest.xml.
50         int id = item.getItemId();
51         if (id == R.id.action_settings) {
52            return true;
53         }
54         return super.onOptionsItemSelected(item);
55      }
56
57      /**
58       * A placeholder fragment containing a simple view.
59       */
60      public static class PlaceholderFragment extends Fragment {
61
62         public PlaceholderFragment() {
63         }
64
65         @Override
66         public View onCreateView(LayoutInflater inflater, ViewGroup container,
67               Bundle savedInstanceState) {
```

```
68          View rootView = inflater.inflate(R.layout.fragment_capture_image,
69                container, false);
70          return rootView;
71      }
72   }
73
74 }
```

4.8.2 Results

Table 4.4 presents the static typestate analysis results for our asynchrony-aware approach on AICCFG against a synchronous-only typestate analysis built as an IFDS over the program representation used by IccTA applied to AsyncBench test applications. On these applications, our asynchrony-aware typestate analysis captures all typestate violations and raises a false warning in only one case in each resource category. We consider these as false warnings, these are due to inherent non-determinism in the control flow interleaving across different component lifecycles. It is hard to reason whether they can actually manifest in some run of the applications. Since these warnings are raised by both asynchrony-aware and asynchrony-unaware approaches, we soundly treat them as false warnings in both the cases. Following this logic, we achieve a precision rate of 78% and a recall rate of 100% over the AsyncBench benchmarks.

Compared to this, the synchronous-only approach misses all the typestate violations in different categories due to the inherent unsoundness of its underlying program representation and its synchronous analysis. The high false negatives of the synchronous only analysis show the unsoundness in the state-of-the-art modeling of the Android environment which makes them miss many typestate violations. Moreover, a sound asynchronous and lifecycle modeling and an asynchrony-aware analysis also increase the precision of our analysis, allowing us to give a 50% lower false warning and higher precision compared to the synchronous-only analysis. Our asynchrony-aware typestate analysis runs smoothly on a normal desktop machine with a dual-core Intel processor and a moderate memory size of 16 GB. The average time for analyzing an application in AsyncBench came out to be approximately 2-3 minutes. This shows the practical feasibility of our analysis on these applications.

4.9 Immediate Future Work

There are several directions for an immediate extensions of the work discussed in this chapter. The formal semantics presented in this thesis excludes the concurrent features of Android applications. Extending the analysis approach to handle multi-threaded control and data flows

Table 4.4: TypeState Analysis on AsyncBench Applications.

⊛ = correct warning, ⊖ = missed violation, ⋆ = false warning		
Application Name	Async-Aware	Sync-only (IccTA)
Camera Applications		
Cameraapp_0	-	⋆
Cameraapp_1	⊛	⊖
Cameraapp_2	⊛	⊖
Cameraapp_3	⊛, ⋆	⊖,⋆
MediaPlayer Applications		
MediaPlayer_0	-	⋆
MediaPlayer_1	⊛	⊖
MediaPlayer_2	⊛	⊖
MediaPlayer_3	⊛, ⋆	⊖,⋆
SQLiteDatabase Applications		
SQLiteDatabaseapp_0	-	⋆
SQLiteDatabaseapp_1	⊛	⊖
SQLiteDatabaseapp_2	⊛	⊖
SQLiteDatabaseapp_3	⊛, ⋆	⊖,⋆
File Applications		
Fileapp_0	-	⋆
Fileapp_1	⊛	⊖
Fileapp_2	⊛	⊖
Fileapp_3	⊛, ⋆	⊖,⋆
UriPermission Applications		
Permissionapp_0	-	⋆
Permissionapp_1	⊛	⊖
Permissionapp_2	⊛	⊖
Total, Precision and Recall		
⊛, higher is better	14	0
⋆, lower is better	4	8
⊖, lower is better	0	14
Precision = ⊛ / (⊛ + ⋆)	78 %	0 %
Recall = ⊛ / (⊛ + ⊖)	100 %	0 %

can be a challenging and interesting extension, as none of the state-of-the-art static analysis works for Android applications handle multi-threading in a sound and practically precise way. Another interesting extension will be to use to model, the AICCFG, and the asynchrony-aware static analysis framework discusses in the chapter to solve other interesting static control and data flow analysis problems, like Information flow analysis, or static slicing, etc.

4.10 Chapter Summary

In this chapter, we motivated for a sound model of Android asynchronous control flow. Following this, we provided such a formal model and its semantics. We presented an intermediate program representation for Android applications, capturing the defined formal model and semantics. We motivated the need for an asynchrony-aware static analysis for Android applications and presented an asynchrony-aware, static typestate analysis over our intermediate program representation. We empirically showed the effectiveness of our model and analysis and compared it against other state-of-the-art analyses.

Chapter 5

Presburger-definable Typestates

The thesis so far discussed the challenges in verifying safety properties over programs with convoluted control and data flow semantics. We also presented solutions for these challenges in the case of Android applications. This allowed us to build a typestate analysis over these applications aiding in statically verifying various important Android resource usage protocols. The AICCFG presented is not limited to typestate analysis and can be used for other static analyses as well, thus allowing us to verify many other program properties like information flow, pointer analysis, etc. In this chapter we discuss the other aspect of program verification which make the safety property verification a challenging task. We discuss about the expressive limitations of the typestate properties and how they affect the verification of some crucial safety properties of programs. Following this we present an extension of classical typestates to overcome some of these expressive limitations.

5.1 Introduction

Types are one of the major mechanisms used by programmers for modeling and verifying properties of programs and data. However, types have limitations as a specification mechanism since the type of an object remains constant during its life time. For instance, types are good at specifying structural properties of data or programs, like type of a method arguments, return value, etc, which generally does nor change during the life time of the object. However, standard types cannot enforce a behavioral property, like calling a method in an allowed order, or availability of a method based on the state of the base object. A classic example of such a property is a FileManager which allows a File object to have a set of operations defined over it, *viz.* open, close, read and write but only a subset of these operations are valid on a File object based on its current state.

Typestate [115, 82, 31, 9, 53] systems, which form a component of general Behavioral types [61], allow the type or some abstract state of an object to change during its lifetime in a computation. Thus, unlike standard types they can enforce behavioral safety properties that depend on the changing state of the object.

Originally, typestate allowed programmers to specify a valid sequence of operations on data, protocols associated with resource usages, behavioral correctness like defining a variable before its use, etc. More recently, typestate has seen its application in newer domains, for example, typestate is useful to specify and model properties and protocols of complex communicating systems [83].

Although efficient at capturing properties related to the state of data, standard typestates again have limited expressive power. Standard typestates can only express program properties from *finite-state* domain [82, 115] and cannot enforce a property beyond the finite state abstraction. For instance, consider a commonly occurring non-regular behavioral property associated with an abstract data type like a *Stack*, "the number of items pushed in the stack is greater than or equal to the number of items popped". Standard typestates cannot enforce such a *context-free* property. This limitation of regular typestate impedes the effectiveness of various typestate analyses, type systems for typestate verification and typestate oriented programming languages approaches for program analysis and verification.

In the lack of a more expressive typestate, such properties are checked using costly, error prone and difficult to debug runtime checks [74] or are converted to abstract models and then verified using existing non-regular verification techniques [15]. These abstract model based verification do not allow correct-by-construction practical implementation of real systems and does not provide any guarantee over a concrete implementation of an abstract protocol. (For example, Heartbleed bug [130] in popular OpenSSL library that provides cryptographic services such as SSL/TLS to the applications and services).

The main idea of our work is to attack this expressive limitation of regular typestates. We present an extended notion of typestate, called Presburger-definable typestate (p-typestate). Presburger-definable typestate allows associating a *Presburger formula* to the states of a regular typestate. This formulation immediately allows a programmer to associate some extended property along with the regular state of an object, this allows expressing richer behavioral properties. However, allowing richer logic shoots up the complexity of typechecking which might lead to an undecidable typechecking [56] or might need user interaction [117]. To strike a balance between expressiveness and decidability and automaton, we use Presburger formulas which allows us to develop a fully automatic decidable static checking system for p-typestate system using the decidable validity and satisfiability results of Presburger logic.

We use the expressive power of *dependent types* [85] to implement the idea of p-typestate and provide a typestate oriented, imperative programming language with p-typestate typesystem called DRIP (**D**ecidable **R**ich **I**nterface **P**rogramming) language. DRIP allows a programmer to specify and construct programs satisfying rich behavioral properties. We present a formal semantics for DRIP and p-typestate system, and present a proof of soundness of our system.

p-typestate and DRIP makes an important contribution to the *design-by-contract* (behavioral *contract*) [84] methodology for object-oriented programs. This methodology allows program contracts and other behavioral properties to be annotated by a programmer with methods which are then asserted during program execution. There are numerous benefits of using such a principled approach for program design, like modular design, better implementation, safety guarantees, etc. However, one of the major hurdles of its acceptance in more main-stream languages has been a lack of *verified contract-by-design systems.*, that is, design-by-contract with mathematical verification that all contracts are always honored. Such a mathematical verification cannot be done using runtime checks and requires a decidable static checking. p-typestate and DRIP provide one such verified *dbc* system using static type-checking. Unlike other *dbc* languages (like Eiffel, Ada, etc.), we do not defer the checking of the implementation satisfying the contract to runtime assertions, and unlike only a few other verified design-by-contract systems like ESC/Java, we have a decidable static type checking.

Further, automatic inductive type checking p-typestate and other such rich behavioral properties usually requires a programmer to annotate loop invariants in the program [126, 99, 112], to placate this burden from the programmer, we also present a novel loop invariant calculation approach using loop acceleration technique for Presburger definable transition system.

Finally, we evaluate our p-typestate system and the static typechecking by implementing programs enforcing several non-regular contracts and properties in DRIP which could not be enforced using regular typestates.

Following are the major contributions of our work-

- We introduce *Presburger-definable typestate*.
- We present a dependent type system to enforce p-typestate properties and implement DRIP which is a typestate-oriented language with decidable static p-typestate checking.
- We present a soundness result for our p-typestate type system over DRIP.
- *Minor contribution* : We present a novel loop-invariant calculation technique based on acceleration techniques for Presburger definable systems.

- We further show the effectiveness of the p-typestates by implementing many real world programs enforcing p-typestate properties in DRIP which cannot be specified using regular typestates.

5.2 Overview

```
1  [  Typestates("_initialized", "_uninitialized", "_empty", "_full", "_working") ]
2  class Stack {
3  // @requires \_initialized == false;
4  // @ensures \_initialized == true;
5  [  Pre(_uninitialized), Post(_initialized AND _empty) ]
6  public Stack(){ return new Stack(); }
7
8  private int capacity;
9  //@ private invariant 0 <= top <= capacity;
10 private int top; // top index
11 private int count;
12
13
14 // @requires \_initialized == true;
15 // @requires 0 < i <= top;
16 // @requires top <= capacity;
17 // @ensures \_full == false;
18 [  Pre(_initialized AND ( _working OR _full), Post(_initialized and ( _working OR _full) ]
19 public get(int i){ ... }
20
21 // @requires _initialized == true;
22 // @requires top < capacity;
23 // @ensures top' > 0;
24 // @ensures top' = top + 1;
25 // @ensures count' = count + 1;
26 // @ensures \_empty == false;
27 [  Pre(_initialized AND (_empty OR _working ), Post(_initialized AND (_working OR _full) ]
28 public put (item x){}
29
30 // @requires \_initialized == true;
31 // @requires \_empty == true;
32 // @requires top <= capacity;
33 // @requires #put == #remove
34 // @ensures \_initialized == false;
35 [  Pre(_initialized AND (_empty ), Post(_uninitialized) ]
36 public balance(){
37    this.discard = true;
38    discardTheStack();}
39   }
40 }
```

Figure 5.1: A typestate based implementation of Stack with contracts

Consider Figure 5.1, an example of an array based *Stack* implementation in an object-oriented setting. The abstract class Stack provides a constructor, 4 methods *put*(Similarly *remove*, which is not shown), *get*, *balance* and a set of private class fields. The figure shows a set of contracts [131, 74] associated with each method in a specification language similar to Java Modeling Language (JML). (Figure also shows a set of typestate annotations as **Pre**

and **Post** annotation withing square brackets, which the reader should ignore for now.) This is a common specification pattern following the *design-by-contract* [84] principle for building reliable softwares. These specifications can then be checked at either compile time or runtime, thus guaranteeing certain properties. For example, the *put* operation *requires* that the stack is *not-full*(the index of the top element is strictly less than the capacity) while the *get* operation requires the index to be in the capacity of the stack and is not defined for an index 0. Such contracts allow a programmer to specify behavioral properties of abstract data or program apart from the structural properties (like type of arguments and return value) which are normally defined by a method signature. Standard typestates can efficiently enforce simple behavioral program properties by associating a finite set of typestates with each type, and annotating each expression with a possible transitions over these typestates in a fashion similar to the JML annotations. Let us see if we can enforce such rich contracts using standard typestates.

Figure also shows a typical typestate annotated implementation for the Stack and related contracts described earlier (we use the same figure for brevity). The annotation language (shown in bold as finite set of **Typestates** and **Pre** and **Post** annotations) has been adapted from the typestate work for objects [82]. We keep the older JML based contracts annotations along with the new typestate annotations for elucidation purposes. We define five different typestates *viz.* { _initialized, _uninitialized, _empty, _full, _working } to capture the contracts described by the contracts. For each method definition, we associate a **Pre** and **Post** clause, showing the analogues *@requires* and *@ensures* annotations of the contract. As can be seen in the figure, for each of these methods, a portion of the original contracts (showing programmers original behavioral intent) could be "efficiently" encoded using typestates while the remaining, either can be encoded only with a complex set of new typestates or cannot be encoded at all. These are shown in red ink in the figure. For example, consider method put at lines (21-28), although standard typestate can encode the simple precondition (@requires _initialized == true) and the postcondition (@ensures _full == false) by using suitable typestates for these states, specifying other contracts shown in red, either requires defining a complex set of typestates (for example @ensures top' > 0 can be specified by defining a new typestate for top and relating it to _empty) or cannot be specified using finite states at all. For example, defining the other two contracts ensuring updated value for top and count requires counting over possibly infinite set of integers and hence cannot be modeled using finite set of typestates. For some other cases, for example method balance at lines (30-39), the contract requires a check (@requires #put == #remove) that the number of items pushed (put) are equal to the number of items popped (remove) which being a non-regular property cannot be expressed and enforced using standard finite state typestates.

A contract specification language like JML, gives a glimpse of useful behavioral contracts or properties which a programmer wishes to be verified/enforced by the language compiler which can aid in simple and correct design and implementation of real world systems. Unfortunately, standard typestate can only enforce a very small fraction (finite state behavioral abstractions) of these useful behavioral properties, as shown by the above example. Furthermore, many of such non-regular behavioral properties show up in varied domains of programs like, API specifications for infinite state systems, cache coherence protocols, session and communication protocols, etc.

In the lack of such a system, these richer non-regular behavioral properties are either dynamically enforced at runtime, which is the approach taken by the JML compiler(jmlc) [132], or are statically enforced using highly expressive but undecidable logics, which is the path taken by the *Extended-static checker* for Java and Modula [56]. Unfortunately, both these approaches have their shortcomings, the runtime assertions which are more popular are costly and error prone, difficult to debug and cannot be verified. The static verification approaches using richer logics unfortunately very soon fall into undecidability trap. For example, ESC/Java although expressive, but lacks a decidable static checking and hence cannot provide any sound guarantees. Other even more richer systems like fully dependently typed languages like Caynene [14] has undecidable typechecking, while other theorem provers like Coq and Agad([117, 98]) require user interactions and are not automatic, making it hard to program real world systems.

To mitigate these limitations, in this work we present an extension of the classical typestates called as *Presburger-definable typestates(p-typestate)* . These typestates allows a programmer to specify non-regular behavioral properties like the one discussed in Figure 5.1 while still maintaining a fully automatic, decidable static checking.

For example, consider Figure 5.2, it shows a p-typestate based implementation of the original Stack and its related contracts in our typestate-oriented programming languag DRIP. The exact syntax and meanings of the notations will be elaborated in Section 5.4. Each method is annotated with a pre and post typestates for its parameters, the base reference field (this) and the fields in the environment. The program defines a *state* Stack rather than a *class*, this is the typestate-oriented intricacy which the reader should ignore at the moment. We still include the *@requires* and *@ensures* annotation of JML contracts for illustration and they are not the part of the program. The initial Typestate annotation (line 1) defines the possible set of regular typestate which will be use to define Presburger definable typestate. In an actual DRIP program, these regular typestates are defined as separate states similar to the Stack. There are a few noteworthy points in this implementation which we discuss now. Line 4, defines a new Presburger typestate called as PStack, which is a dependent function type dependent on

```
1  [  Typestates("_initialized", "_uninitialized") ]
2  state Stack {
3      // a dependent function type with \#p , \#r a formula over it and a possible regular typestate.
4      type PStack : Pi (  n_p, n_r, n_p ≥ n_r  ) -> Stack;
5      Item[] array  = new Item[capacity];
6      var Integer⟨n_cap⟩  capacity;
7      var Integer⟨n_top⟩  top; // top index
8      var Integer⟨n_count⟩  count;
9
10 // @requires \_initialized == false;
11 // @ensures \_initialized == true;
12 public Stack()[PStack ( True ) -> \_uninitialized >> PStack ( n_p == 0, n_r == 0
       ) -> \_initialized  this]{
13    return unique PStack (0, 0) -> \_initialized st = new Stack();
14 }
15
16   //@requires \_initialized == true;
17   //@requires 0 < i <= top;
18   //@requires top <= capacity;
19   //@ensures _full == false;
20 public get(Integer ⟨n_i⟩ i)[ PStack (_) -> \_initialized >> PStack (_) -> \_initialized this , Integer
       ⟨n_top, 0 < n_i ≤ n_top⟩ top  , Integer⟨n_cap, n_top ≤ n_cap⟩  >> Integer⟨n_top ≤ n_cap⟩
        capacity ]
21 {
22   //...
23   return array[i];}
24
25   // @requires \_initialized == true;
26   // @requires top < capacity;
27   // @ensures top' > 0;
28   // @ensures top' = top + 1;
29   // @ensures count' = count + 1;
30   // @ensures _empty == false;
31 public put (Item x)[ PStack (n_p, n_r) -> _initialized >> PStack
       (n'_p = n_p + 1, n_r) -> \_initialized this, Integer⟨n_top, 0 ≤ n_top < n_capacity⟩ >> Integer⟨n'_top = n_top + 1⟩ top
       ,
32 Integer⟨n_count⟩ >> Integer⟨n'_count = n_count + 1⟩ ]{
33     top = top + 1;
34     array[top] = x;
35     count = count + 1;
36     this <- PStack (n'_p = n_p + 1, n_r) -> \_initialized;
37 }
38
39   // @requires \_initialized == true;
40   // @requires \_empty == true;
41   // @requires top <= capacity;
42   // @requires #put == #remove
43   // @ensures \_initialized == false;
44 public balance()[PStack (n_p, n_r, n_r == n_p) -> _initialized >> PStack
       (n'_p = n_p, n'_r = n_r) -> \_uninitialized this, Integer⟨n_top ≤ n_capacity⟩ top ]{
45    this.discard = true;
46    discardTheStack();
47
48 }
```

Figure 5.2: A p-typestate based implementation of Stack(PStack) with contracts

a Presburger formula over two auxiliary variables (n_p, n_r) counting the abstract properties of the Stack, *viz.* #p and #r. The dependent function type also depends on a regular typestate state (Stack) which can be either _initialized or _uninitialized.

Besides these p-typestate types, the program contains simple Java types and declarations (line 5) and dependent version of *int*. These are represented as Integer $\langle n_{cap} \rangle$ capacity, representing an *int* capacity dependent on its value n_{cap}. Let us see how these simple, dependent and p-typestate types allow a programmer to specify and verify earlier JML contracts which could not be specified using regular typestate (earlier shown in red, now shown in green). Consider the definition of method put, the method signature defines the pre and the post p-typestates (and types) for all the relevant fields using a syntax $A \gg A'\ a$, where A and A' are pre and post types for the variable a. The signature *requires* that the base object is a PStack, dependent on n_p, n_r counting #p and #r and is in an _initialized state (represented as, PStack (n_p, n_r) $\rightarrow$ _initialized) and post the method execution, it *ensures* that the #p is incremented and the state of the PStack remains _initialized. Similar contracts are defined for other variables in the environment, like top, capacity, count and for method parameters as well (not shown here). Similarly, the signature for method balance defines a non-regular contract which standard typestate cannot express. Readers can note, that some of the auxiliary variables (used to define Presburger formula terms) have their analogues as program fields, like n_{top} has top and n_{count} has count, these variables model some abstract state of the data/program required to verify the property and contract. These are updated by our static typechecker implicitly and programmer does not need to track them. Contrary to this, some other auxiliary variables (e.g. n_p and n_r) model some extended abstract property of the data/program not captured by the program fields, like the number of *put* and *remove* operations performed on a Stack. These are only visible to the typecheker to specify and verify an extended abstract property of the program/data. These need to be explicitly updated by the programmer, for example line 36 shows such an explicit update of the p-typestate for the *this* field. The semantics of an implicit/explicit update will be discussed in detail in Section 5.4.

Alongside statically verifying protocol implementations, DRIP also allows a programmer to verify a correct usage for such rich contracts and interfaces. For example, consider Figure 5.3, which is a faulty BoundedStack client program, which is adapted from [86]. The figure presents two different tests t1, t2 with main methods. Each main has a sequence of DRIP statements and expressions including a loop. Each of the tests, uses the p-typestate implementation of Stack from Figure 5.2 and the typechecker can statically verify the bug in t2 while verifying the correctness of t1. The while loop contains an invariant (shown in [...]), which is checked by the DRIP typesystem and then used for inductively verifying other expressions. Later we also

discuss, how such loop-invariants can be automatically calculated thus alleviating the burden from a programmer.

Thus, p-typestate and DRIP provides a verified design-by-contract system and a programming language which allows a programmer to implement correct by construction programs guaranteeing rich behavioral properties which cannot be enforced using standard typestates. Further, the static p-typestate checking does not fall into the undecidable trap like other verified design-by-contract systems like ESC/Java, thus providing safe guarantees for an interface implementation and usage. s In next few sections, we present a formal definition of the p-typestate and DRIP and details of the typechecking.

```
1  // @Test t1
2  import Stack;
3  public main(){
4    Stack st = new Stack();
5    st.capacity = 5;
6    st.put(2); st.put(3);
7    st.get(1); st.remove(); st.remove();
8    st.put(8); st.remove();
9    Integer ⟨n_i⟩ i = 1;
10   while [st.count <= st.capacity](i <
         capacity ){
11     st.put(i);
12   }
13
14   st.put(2); //  typechecks
15   st.remove(); st.remove();
16   st.remove(); st.remove();
17   st.remove(); st.balance(); // typechecks
18   st = new Stack();
19
20 }
```

```
1  // @Test t2
2  import Stack;
3  public main(){
4    Stack st = new Stack();
5    st.capacity = 5;
6    st.put(2); st.put(3);
7    st.get(1); st.remove();
8    st.remove(); st.put(8);
9    st.put(4) st.remove();
10   Integer <@⟨n_i⟩@> i = 1;
11   while [st.count <= st.capacity](i <
         capacity ){ // p-typestate error
12     st.put(i);
13   }
14
15   st.balance(); // p-tyepsate error
16   st = new Stack();
17
18 }
```

Figure 5.3: Two test applications using DRIP PStack

5.3 Presburger-definable Typestate

In this section we discuss the main idea of Presburger-definable typestate, present a formal definition of the concept and discuss how it can be modeled in practical systems.

5.3.1 Formal Definitions

Definition 5.1 (Typestate [115]) Given a strongly typed language $\mathcal{L}$ and an extensible set of types $\mathcal{T}$, the typestate (or regular typestate) associated with a type $\tau \in \mathcal{T}$ (or a variable of type τ), is a *finite set* of static *labels* $S(\tau)$.

For example, the regular typestate S(File) for the simple File type for a FileManager property is a set $\{Open, Close, \perp\}$. Intuitively the typestate set defines the finite set of possible states for

a variable of type τ.

Definition 5.2 (Presburger-definable Typestate) A Presburger-definable typestate (p-typestate) associated with a type $\tau \in \mathcal{T}$ (or a variable of type τ) is a possibly infinite set $S_p(\tau) \subseteq (\Psi \times S(\tau))$, where Ψ is the set of Presburger formulas defined over a set of auxiliary integer variables called type-variables, and set $S(\tau)$ is the regular typestate set for τ.

For example, the p-typestate set for type File can be defined as follows. Let $S(File) = \{open, close, \perp\}$ be the regular typestate set for File, and let $\phi_1 = \{\forall i, j | i >= j\}$ and $\phi_2 = \{\forall x, y | x = y+1\}$. Then pairs like ($\phi_1$, *open*) and ($\phi_2$, *close*)) $\in S_p(\tau)$. Intuitively, a p-typestate allows to associate a presburger definable property along with a state, thereby providing greater expressiveness than a typestate.

The set of integer variables like $i, j, x, y, ...$ can be either *programmable*, which can be explicitly updated by a programmer or *non-programmable*, which are operated upon only by the typechecker. The latter are useful for modeling some abstract state of programs/data which is a function of a set of program fields. For example, a File might have an integer field *wordcount*, which counts the number of words written to the *File*. In such a case a *File* type can capture the *wordcount* with an auxiliary variable, say n_{wc}. This relation between *wordcount* and its associated n_{wc} is maintained by the typechecker. These variables are like *Model fields* of JML. Unlike these, programmable auxiliary integer variables are introduced by a programmer to model some extended state of programs/data. For example, in the Stack example of Figure 5.2, auxiliary variables n_p, n_r have no associated program variables and are used to model an extended property capturing number of items pushed and removed respectively. These variables can be explicitly updated by the programmer in our DRIP. These variables are like *Ghost fields* of JML. These programmable and non-programmable integer variables form *type variables* in our language. The details of these variables and allowed operations on them will be described in section 5.4.

5.3.1.1 p-typestate Transitions

To define transitions over p-typestates, we need to define the set of program operations. We define these operations here in a simple way. For a strongly typed language $\mathcal{L}$, we formally define a program $\mathbb{P}$ in $\mathcal{L}$. A program $\mathbb{P}$ is a sequence of expressions where each expression is defined as a pair $< op, \overline{V} >$, *op* being an operation from the set of valid operations $\mathcal{O}$ and $\overline{V} =< v_1, v_2, ..., v_N >$ is an indexed set of operands to *op*. Each operation $op \in \mathcal{O}$ has a signature $\mathcal{T}(op) = \langle t_1 : \tau_1, t_2 : \tau_2, ..., t_n : \tau_n, t_r : \tau_r \rangle$, specifying the type of its operands with the last term representing the result. Each τ_i, represents the type of the i_{th} operand term t_i. Each

v_i is an actual argument for the formal operand with the restriction on type of v_i, $Typeof(v_i)$ = τ_i. We define p-typestate transitions ($\delta : S_\Psi(\tau) \mapsto 2^{S_\Psi(\tau)}$) for each operation $op \in O$ as ($Pre_{op,i} \gg \{Post_{op,i,k}\}$), such that-

- $Pre_{op,i}$, is the p-typestate precondition for the i_{th} operand v_i, $Pre_{op,i} \in \mathcal{S}_\Psi(Typeof(v_i))$, defines the required p-typestate for v_i for *op* to be applicable.
- For each different outcome k, k = 1, 2, ..., m, $Post_{op,i,k} \in \mathcal{S}_\Psi(Typeof(v_i))$ represents the typestate for v_i, when *op* terminates with outcome k.

5.3.1.2 Capturing p-typestate with Dependent Types

Almost all the static standard typestate analysis and verification works([53, 9, 115]) model typestates as a set of static labels, captured using some finite set of standard types, and define semantics for these types and their transitions. For example, the typestates associated with the *FileManager* example can be modeled as a set of static types Open, Close and Error, and the transitions of the system can be modeled as simple rules over method definitions for *open, close, read, etc.*.

Unfortunately, this approach is not directly extensible to capture definitions and transitions of a Presburger-definable typestate. The problem arises due to the possible set of p-typestates forming an infinite set. This makes statically defining a predefined set of simple static types impossible. Furthermore, a p-typestate transition can possibly require counting and comparing integer values in form of Presburger formulas for which simple types do not suffice. Thus, the approach fails to specify p-typestate transitions when trying to model them as regular typestate transitions. Specifying p-tyepstates and their transitions and checking a program against these specifications requires a richer theory which can model possibly infinite p-typestate states and operations over an integer counter system.

Dependent Types [88, 22] are types which can depend on some other terms in a program. Unfortunately, this expressiveness of general dependent type theories like Martin Löf's type theory (theory behind Coq and Agda) comes at the cost of complex type-checking (undecidable typechecking in general). We choose a fragment of these theories which is expressive enough to define above stated features of p-typestate, and yet has a fully automatic, decidable typechecking. This fragment restricts the *index terms* of a dependent type to a product of Presburger arithmetic logical formulas and finite set of regular typestate states.

Using this fragment, a programmer can create a possibly infinite family of types (modeling the set of p-typestate states) as a *dependent function* of a pair (x,s) $\in (\Psi \times S)$. An element of this set can be constructed by a standard *dependent function application*. For example, let

$\Pi(x : \phi, s : S, p^{x,s})$ be a dependent function type, indexed by a pair (x, s) of Presburger formula and a state. let $z = \forall i_1, i_2.(i_1 = 3 \wedge i_2 \geq i_1)$ be a valid Presburger formula and let s_1 be a valid regular typestate state, then the dependent function type application gives a concrete p-typestate state, represented by $p^{x,s}\{z/x, s_1/s\}$. Such a state can be pictorially represent as follows:

$$(\forall i_1, i_2.(i_1 = 3 \wedge i_2 \geq i_1))$$
$$(s)$$

A p-typestate transition over these p-typestate states is defined by pre- and post- method annotations over these states (types constructed via dependent function applications) while a dependent typechecking algorithm enforces a given p-typestate property over programs.

5.4 Decidable Rich Interface Programming (DRIP) Language and p-typestate Type System

In this section, we present a formal definition of our dependent type system implementing p-typestate. We also present a small, typestate-oriented, imperative language with monomorphic dependent types incorporating the p-typestate and static typechecking and call it Decidable Rich Interface Programming Language (DRIP).

5.4.1 Syntax

Abstract Syntax Table 5.1, presents an abstract syntax for DRIP. The syntax is inspired by a standard object-oriented language or a core object calculus like Feather Weight Java [65]. Besides the aspect of typestates and Presburger-definable typestates, the language is similar to a standard object-oriented language. The language has *states* as first-class elements (following typestate-oriented programming [9]) rather than the usual classes. This allows a programmer to define typestate properties explicitly in the program rather than as a meta property associated with a class.

For illustration the syntactic constructs specifically relevant to the p-typestate definitions are shown in gray background.

Before explaining the syntactic features of the language in detail, we elaborate the names used in the syntax. $S, S_1, S_2...$ and $s_1, s_2, s_3, ...$ represent regular state names. $f, g, h..$ range over field names while $M, N, ...$ and $m, n, ...$ range over method names. $x, y, ...$ range over variables, and $\rho_1, \rho_2, ...$ range over names of general variables ranging both over values and references and type variables. $\overline{X}$, represent a possibly empty list of an element X. A semicolon (;) represents

(program)	(P)	::=	$st_1\ st_2\ st_3\ ...\ st_n\ main$
(state)	(st)	::=	state S_1 **case of** S_2 { $\overline{\text{d}}$ }
(declaration)	(d)	::=	field | method | pts-def
(field-decl)	(field)	::=	τ f
(method-decl)	(method)	::=	$M : \chi$ { b }
(signature)	(χ)	::=	$\forall[\nabla]\ (\rho_1 : \tau_1, \rho_2 : \tau_2, ..., \rho_n : \tau_n)$ $\rightarrow \exists \rho_r : \tau_r;\ (\rho_1 : \tau'_1, \rho_2 : \tau'_2, ..., \rho_n : \tau'_n)$
(type-decl)	(pts-def)	::=	type τ
(code-block)	(b)	::=	$\overline{stmt}$
(statement)	(stmt)	::=	let x = e in stmt
	(state-change)		| e $\leftarrow$ e in stmt
			| while $[\exists.\phi]$ (e_1 : bool, e_2) in stmt
			| if (e : bool) then e_1 else e_2 in stmt
			| skip
(expression)	(e)	::=	x | new S | e.f
			| e.m($e_1, e_2, ..., e_p$)
	(comp)		| e ; e | c
	(dep-fun-app)		| $p^{x,s}$ (e_2)
	(pair)		| ($e_1 : \tau_1$, $e_2 : \tau_2$)
	(projection)		| pr_1 pair | pr_2 pair
	(value)		| v
(const)	(c)	::=	boolliteral | intliteral | stringliteral
(value)	v	::=	ρ | c | new S | (e_1, e_2)
(name env)	(∇)	::=	$\bullet$ | ρ, ∇

(variable name)	x , this
(field name)	f
(method name)	m , main, M
(type family name)	γ
(state name)	S
(abstract locations)	l_i
(arithmetic variable name)	v

Table 5.1: Abstract Syntax declarations, statements, expressions, and values

end marker and is omitted from statements and expressions (other than wherever required for explanation) for clarity.

A syntactically valid program (P) (refer Table 5.1) is a concatenation of zero or more state definitions (st_1, st_2, ..., st_n) followed by an optional method declaration for a *main* method. A *state* models a state of a regular typestate. A state definition (st) is similar to a class declaration in a standard object calculus. It defines a state S_1, which may be a *substate* (like subclass) of another state S_2 and has a list of declarations d. The finite set of state definitions $\{ st_1, st_2, ..., st_n \}$ in a program defines the possible set of regular typestates for a given p-typestate property.

A declaration (d) can be either a *field declaration* (field), a *method declaration* (method), or a *dependent function type* declaration (pts-def). DRIP allows a programmer to define a new *dependent function type* using the *pts-def* syntax. It declares a new dependent-function type which is dependent on a Presburger definable formula, ϕ and a regular typestate state S. Intuitively this can be seen as a function which takes a pair (x, s) of these types and returns a type $p^{x,s}$, where x and s can occur free. The resultant type defines a p-typestate state satisfying a presburger formulas x with a regular typestate set s.

A method declaration has a method name M, a method signature χ and a method body b. A method signature is named χ and represents a set of pre- and post- p-typestate conditions for the method. It shows all possible transition of typestates (p-typestate as well as regular typestate) associated with each parameter and environment variable for the method before and after the method execution. A method signature χ looks as follows:

$$\forall[\nabla](\rho_0 : \tau_0, \rho_2 : \tau_2, ..., \rho_n : \tau_n) \rightarrow \exists \rho_r : \tau_r; (\rho_0 : \tau_0', \rho_2 : \tau_2', ..., \rho_n : \tau_n')$$

Where, ∇ is the list of variable names ρ which can be either a value variable which can be assigned a value or a reference variable referring to an object or a programmable type variable. ρ_i is i_{th} argument to the method, with ρ_0 being the base object of the method. The primed version of each of these variables represents their value at the method exit. A method signature expresses possible p-typestate transitions resulting from the execution of a method.

In the concrete language syntax, a method declaration is represented as follows-

$$\tau_r \; m_i(\overline{\tau_{ai} \gg \tau_{ai'} a_i})[\overline{\tau_j \gg \tau_{j'} a_j}]\{field; stmt; e\}$$

Where each p-typestate pre and post annotation (as shown in abstract syntax of χ) for a variable a_j is represented as $\tau_j \gg \tau_{j'} a_j$. The bounded arguments (shown in "( )") and the free variables (shown in "[ ]") are separated out. For instance, consider method definition for put,

defined in PStack state in Figure 5.2 (line 38). The method signature is defined as follows-

$$[PStack(n_p, n_r) \rightarrow \_initialized \gg PStack(n'_p = n_p + 1, n_r) \rightarrow \_initialized\ this,$$
$$Integer\langle n_{top},\ 0 \leq n_{top} < n_{capacity}\rangle \gg Integer\langle n'_{top} = n_{top} + 1\rangle\ top,$$
$$Integer\langle n_{count}\rangle \gg Integer\langle n'_{count} = n_{count} + 1\rangle\ count] \quad (5.1)$$

The signature has no pre- or post conditions for the parameter of the method (as the p-typestate contract "()" is empty), while the method puts a few pre and post contracts on the base object (*this*) as well as other variables in the environment. A p-typesate contract is defined as a " pre $\gg$ post " string. For example, the contract for *this* field requires the base object to be a of type $PStack(n_p, n_r) \rightarrow \_initialized$, which expresses that it is a function of two type variables and must be *_initialized*, for a valid method call, while the post contract $PStack(n'_p = n_p + 1, n_r) \rightarrow \_initialized$ shows how the type variables and the state changes upon the successful execution of the *put* method. The contracts defined for other environment variables, *viz. count, top* can be understood in a similar fashion.

A method body (b) is a list of *statement* declarations. The language *statements* include a few unique statements along with a set of standard object-oriented language statements. The addition includes, an explicit *state-change* statement, a *while* statement with annotated loop-invariant, a skip statement, etc. The *state-change* allows a programmer to explicitly update the state of an expression or a field (having a valuation for programmable type variables and regular typestate) with a new state (with a new valuation for programmable type variables and regular typestate). This is similar to an *assume* statement [87] common in systems for Hoare style program verification. The explicit state-change assumes that the programmer guarantees a new state assigned to the left-hand expression and the typechecker can safely use this assumption while checking later statements and expressions.

The language has a standard *conditional* statement (the implementation also includes the standard *switch-case* statements). The *while* statement is similar to a standard imperative *while* with a small addition. The statement allows a programmer to augment a Presburger formula $\exists.\phi$ as an invariant for the loop. Such a loop invariant is a requirement for the termination guarantees of our typechecker. Unfortunately, this burden on the programmer is a major bottleneck for almost all static checkers over rich type theories. In Section 5.6.3, we discuss how we placate this burden in DRIP, the details are outside the scope of the current work.

An expression e is either a variable x,(both scalar and reference), a *new* expression, which instantiates a *state*, a field access, a virtual method call similar to a standard object based

language (the language assumes all methods calls are virtual, the constructor is called on a new object), a *comp*, composing two expressions, a *constant* c or a *value* v. Besides these standard expressions, the language also has a dependent function application expression, *app* and a simple pair construction expression *pair*. The former allows a programmer to construct a new p-typestate state while the latter allows creating a pair to be passed as an argument to a dependent function type.

5.4.2 Types

(simple type)	(τ)	::=	void | int | bool | string
(state)			| S
(Presburger Formula)			| ϕ
(dependent function)			| $\Pi(z : \phi, s : S).\tau$ | $\Pi(n : \mathbb{N}).\tau$
(dependent integer)			| $(p^{\{x,s\}})$ | Integer $\langle n \rangle$
(pair type)			| pair$(x : \tau_1,\ y : \tau_2)$
(p-typestate)			| $(p^{x,s}(x_1/x, s_1/s))$
(permission type)	(product)		| (a, τ)
(permission)	(a)	::=	unique | immutable
(type context)	(Γ)	::=	$\bullet$ | δ, Γ
(type map)	(δ)	::=	x : τ | e : τ | M : τ | P : τ | f : τ | τ : Type
(Presburger Formula)	ϕ	::=	b | $\phi_1 \wedge \phi_2$ | $\phi_1 \vee \phi_2$ | $\sim \phi$ | $\exists v.\phi$
(Boolean Expression	(b)	::=	true | false | $i \# j$ st $\# \in \{\leq, \geq, \neq, ==\}$
(Arithmetic Expression)	(i)	::=	constant | v | constant * a | $i_1 + i_2$ | - i
(Index Context)	(Φ)	::=	$\bullet$ | ϕ, Φ

Table 5.2: DRIP Types and Context

DRIP has a succinct yet expressive set of types. The major addition to other typestate-oriented programming languages and type systems is the addition of dependent function type(s) to define a family of p-typestates, a type defining Presburger formulas and a *pair* type. A simple type τ can be a primitive type like *int, bool, string, void*, which defines type for each of the constants *intliteral, boolliteral, stringliteral* and a type for *skip* expression respectively. A state S is a valid user-defined type. A well-formed Presburger formula ϕ is a valid type. A *dependent function* $\Pi(z : \phi, s : S.p^{z,s})$ is a valid type. Unlike general dependent types, a dependent function type in DRIP has a restricted domain, it can depend on a pair of, a Presburger formula z, and a regular typestate state s. Another flavor of a dependent function type is $\Pi(n : \mathbb{N}.Integer\langle n \rangle)$ which is a flavored version (a specific case) of the general dependent function type, defining an $Integer$, whose value is dependent on the value of a type variable n. The type variable in this case can again be either *programmable* or *non-programmable*. However, in most practical cases,

the type variable of this special type is non-programmable, and it has a direct correlation with an integer program variable, hence its value can be updated only by the typechecker. A *pair* type is a standard pair of two typed expressions. A *type context* is either empty or has type maps from variables, expressions, declarations, etc. to types. We assume a binding $\tau : Type$ for each valid type τ in the language. A Presburger formula has a standard definition of *First Order* logical constraints with arithmetic addition and constant multiplication over auxiliary integer variables. Finally, the typing rules also maintain an Index Context Φ, keeping track of Presburger definable constraints generated via various typing rules. The Index Context is either empty or a list of Presburger formulas.

5.4.3 Static Semantics of DRIP

The abstract static state of the program is defined as a tuple (Φ, Γ) representing the Presburger formula defined *index-environment* Φ and a type context Γ. The index-environment captures the conjunction of Presburger formulas defined over programmable and non-programmable type variables. The static semantics defines the type and p-typestate safety rules for the language. Figures 5.4, 5.5 and 5.7 present static typing rules for *dependent-types*, *statements* and *expressions*. A type judgment is either of the following forms:

$$(\Phi, \Gamma) \vdash e : \tau \dashv (\Phi', \Gamma') \mid (\Phi, \Gamma) \vdash stmt \dashv (\Phi', \Gamma')$$

The former judgment form says, that under an incoming *index-environment* Φ and an incoming *type-context* Γ, an expression e is evaluated to a type τ while updating the *index-environment* and *type-context* to Φ' and Γ' respectively. The latter form says, that a *stmt* is well-typed in the static state, and typechecking the *stmt* updates the *index-environment* and *type-context* to Φ' and Γ' respectively. We leave out the later $(\dashv (\Phi', \Gamma'))$ from a judgment if $(\Phi' = \Phi)$ and $(\Gamma' = \Gamma; e : \tau)$ to reduce cluttering.

It is easy to see how these *index-environment* and *type-context* might get updated during the typechecking process. For example, a new *dependent function type* $(\Pi(x : \phi, s : S, p^{x,s}))$ declaration using Rule (T-Pi-I) updates the incoming Index environment Φ to a new environment $\Phi \wedge x$ which is further extended with $\Omega(s)$, which is a method which returns a Presburger formula representing the state s. A well-formed or a valid type has a higher type Type.

5.4.3.1 Type System and Typing Rules

The p-typestate type system and corresponding typing rules enforce a p-typestate property. Many typing rules for DRIP are similar to a strongly-typed object-based language, thus for clarity, we only discuss rules which are unique to p-typestate system.

$$\boxed{(\Phi, \Gamma) \vdash e : \tau \dashv (\Phi', \Gamma') \mid (\Phi, \Gamma) \vdash stmt \dashv (\Phi', \Gamma') \mid (\Phi, \Gamma) \vdash e : \tau}$$

$$\text{T-Pi-F}\frac{\Phi, \Gamma \vdash \phi : \text{Type} \qquad \Phi, \Gamma \vdash S : \text{Type}}{\Gamma, \Phi \vdash \Pi(x : \phi, s : S).\tau : \text{Type}} \qquad \text{T-pair-F}\frac{\Phi, \Gamma \vdash \tau_1 : \text{Type} \qquad \Phi, \Gamma \vdash \tau_2 : \text{Type}}{\Phi, \Gamma \vdash pair(x : \tau_1, y : \tau_2) : \text{Type}}$$

$$\text{T-Pi-I}\frac{\Phi, \Gamma \vdash x : \phi \qquad \Phi, \Gamma \vdash s : S}{\Phi, \Gamma \vdash (\text{type Pi}(x : \phi, s : S).p^{x,s}) : \Pi(x : \phi, s : S).\tau) \dashv \Phi \wedge x \wedge \Omega(s), \Gamma}$$

$$\text{T-pair-I}\frac{\Phi, \Gamma \vdash x : \tau_1 \qquad \Phi, \Gamma \vdash y : \tau_2}{\Phi, \Gamma \vdash (x, y) : pair(x : \tau_1, y : \tau_2)}$$

$$\text{T-pair-pr1}\frac{\Phi, \Gamma \vdash (x, y) : pair(x : \tau_1, y : \tau_2)}{\Phi, \Gamma \vdash pr_1(x, y) : \tau_1}$$

$$\text{T-pair-pr2}\frac{\Phi, \Gamma \vdash (x, y) : pair(x : \tau_1, y : \tau_2)}{\Phi, \Gamma \vdash pr_2(x, y) : \tau_2}$$

Figure 5.4: Dependent Function and Pair Type Formation and Introduction Rules

$$\text{T-pair}\frac{\Phi, \Gamma \vdash x : \tau_1 \qquad \Phi, \Gamma \vdash y : \tau_2}{\Phi, \Gamma \vdash (x, y) : pair(x : \tau_1, y : \tau_2)}$$

$$\text{T-dep-app}\frac{\Phi, \Gamma \vdash \overbrace{\text{type Pi}(x : \phi, s : S).p^{x,s}}^{p^{x,s}} : \Pi(x : \phi, s : S).\tau \qquad \Phi, \Gamma \vdash e_2 : pair(x_1 : \phi, x_2 : S)}{\Phi, \Gamma \vdash p^{x,s}\ (e_2) : p^{x,s}\{pr_1(e_2)/x, pr_2(e_2)/s\} \dashv \Phi \wedge pr_1(e_2) \wedge \Omega(pr_2(e_2)), \Gamma}$$

Figure 5.5: Expression Typing Rules

Figure 5.4 presents typing rules for *formation* and *introduction* of *dependent function* type and *pair* type. Dependent function type formation rule (T-Pi-F) defines the formation of a valid user defined dependent function type in DRIP. The rule restricts the allowed class of dependent function type in the language. Thus, a programmer can define a dependent function type, dependent on a term x of a Presburger formula type ϕ, and a valid state s of the State type but not a general dependent function type, dependent on a term of any general type τ in the program. Rule (T-Pi-I), is the dependent function type introduction rule. It presents the typing rule for a dependent function type constructor defined by a type declaration term (pts-def). The rule introduces a new dependent function type *type* $(x : \phi, s : S, p^{x,s})$, which we write in brief as $p^{x,s}$. It checks that the type is dependent on a term x of type Presburger formula, ϕ, and a term s of user-defined regular typestate state S. The rule introduces two

$$\text{T-}p^{x,s}\text{-eq}\frac{\Phi, \Gamma \vdash x_1 = x_2 \qquad \Phi, \Gamma \vdash s_1 = s_2}{\Phi, \Gamma \vdash p^{x,s}\{x_1/x, s_1/s\} = p^{x,s}\{x_2/x, s_2/s\}}$$

Figure 5.6: Dependent Function Application Equality

new Presburger formulas (using the indexed formula x and the index state s) into the *index-environment* thus updating the incoming Φ to a new environment $\Phi \wedge x \wedge \Omega(s)$. Practically, each dependent function type $p^{x,s}$ which is introduced through this rule, represents a family of *p-typestate states*, indexed by a Presburger formula and a regular typestate state. For example, consider Figure 5.2, line 4 represents a p-typestate introduction PStack(Pi($n_p, n_r, n_p \geq n_r$) $\rightarrow$ Stack). This is a scary looking syntax for $p^{(\exists . n_p, n_r, n_p \geq n_r, \exists s.s <: Stack)}$. This is a dependent function type, which when applied to a pair $\langle\ (\exists n_p, n_r . n_p = 2, n_r = 1), (\exists s.s = \_initialized)\rangle$ returns a type representing a p-typestate state of the Stack. The formation rule for a pair type, (T-pair-F) rule defines a valid pair type. Unlike, a dependent function type, a pair type is not restricted to a particular set of types and a programmer can define a pair type for any two types τ_1, τ_2. The introduction Rule T-pair-I, defines a typing rule for a pair type constructor *(x, y)*. A pair type has two predefined projection functions pr_1 and pr_2. These functions take a pair type (x, y) and return its first and second component respectively. Rules (T-pair-pr1) and (T-pair-pr2) define typing rules for these projection functions.

Expressions and statements typing Figures 5.5 and 5.7, present typing rules for expressions and statements.

Rule (T-pair) is an introduction rule which defines typing rule for a *pair* constructor (x, y). Rule (T-dep-app) is interesting, it defines the rule for a dependent function application for the application expression $p^{x,s}\ (e_2)$. It ensures that expression e_2 is a pair of a Presburger formula and a State. It finally reduces the expression $p^{x,s}\ (e_2)$ and assigns it a type $p^{x,s}\{pr_1(e_2)/x, pr_2(e_2)/s\}$. It is a type where free occurrences of x in the result type $p^{x,s}$ are replaced with first projection of the pair and free occurrences of s are replaced with the second projection of the pair. The rule also introduces two new Presburger formulas into the incoming *index-environment* Φ. The first formula is the first projection of the $(pr_1(e_2), pr_2(e_2))$ passed as an argument to the function. The second formulas represented as $\Omega(pr_2(e_2)$, which is a Presburger formula representation of the regular typestate state x_2. Although, the syntax of DRIP allows a programmer to explicitly attach a state (such as s_i) in a p-typestate which allows her an abstraction to define more realistic properties, the typechecker implicitly converts this syntax to a separate Presburger formula (such as $x_i == s_i$, where x_i is a fresh type variable). This allows a cleaner environment and easier constraint solving. $\Omega(pr_2(e_2)$, represent

$$\text{T-invoke}\frac{\begin{array}{c}\Phi,\Gamma \vdash e : S_b \mid e : p^{\{x,S_b\}} \dashv \Phi_1,\Gamma_1 \\ \Gamma(S_b :: m) = m : \Psi\{\bar{b}\} \\ \Psi = \forall[\nabla](\rho_1 : \tau_1, ..., \rho_n : \tau_n, \rho_{n+1} : \tau_{n+1}, ..., \rho_p : \tau_p) \\ \rightarrow \exists \rho_r : \tau_r \ (\rho'_1 : \tau'_1, ..., \rho'_n : \tau'_n, \rho'_{n+1} : \tau'_{n+1}, ..., \rho'_p : \tau'_p) \\ \Phi_1,\Gamma_1 \vdash e_1 : \tau_1 \dashv \Phi_2,\Gamma_2 \\ \Phi_2,\Gamma_2 \vdash e_2 : \tau_2 \dashv \Phi_3,\Gamma_3 \\ ... \\ \Phi_n,\Gamma_n \vdash e_n : \tau_n \dashv \Phi_{n+1},\Gamma_{n+1} \\ \Gamma_{n+1}(\rho_{n+1}) : \tau_{n+1} \\ ... \\ \Gamma_{n+1}(\rho_p) : \tau_p \dashv \Phi'',\Gamma'' \\ \forall\ i \in [1...n].\Gamma''[(e_i) \mapsto \tau'_i] \\ \forall\ j \in [n+1...p].\Gamma''[\rho_j \mapsto \tau'_j]\end{array}}{\Phi,\Gamma \vdash e.m(e_1, e_2, ...e_n) : \tau_r \dashv \Phi'',\Gamma''}$$

$$\text{T-let-1}\frac{\Phi,\Gamma \vdash \text{e} : \tau_1 \dashv \Phi',\Gamma' \quad (\Gamma(x) \neq Integer\langle n_x\rangle) \quad \Phi',\Gamma';\text{x} : \tau_1 \vdash stmt\{\text{x/e}\} : \tau \dashv \Phi'',\Gamma''}{\Phi,\Gamma \vdash \text{let x = e in stmt} : \tau \dashv \Phi'',\Gamma''}$$

$$\text{T-let-2}\frac{\begin{array}{c}(\Gamma(x) = Integer\langle n_x\rangle \quad n_x \in NP(\Phi) \\ v_{app} = Apply(\text{v}) \\ \Phi,\Gamma \vdash stmt\{\text{x}/v_{app}\} : \tau \dashv \Phi',\Gamma'\end{array}}{\Phi,\Gamma \vdash \text{let x = v in stmt} : \tau \dashv \Phi' \wedge \{n_x == v_{app}\},\Gamma'}$$

$$\text{T-update}\frac{\Gamma(e_2) = p^{\phi_2,s_2} \quad \Gamma(e_1) = p^{\phi_1,s_1} \quad \exists x_i, (x_i == s_1) \in \Phi}{\Phi,\Gamma \vdash e_1 \leftarrow e_2 \dashv \Phi \wedge \phi_2 \wedge (x_i == s_2),\Gamma; e_1 : p^{\phi_2,s_2}}$$

$$\text{T-while}\frac{\begin{array}{c}\Phi,\Gamma \vdash \varphi : \phi \qquad \Phi,\Gamma \vdash e_1 : bool \dashv \Phi',\Gamma' \\ (\Phi'; e_1 == true; \varphi),\Gamma' \vdash e_2 : \tau_2 \dashv \Phi'',\Gamma'' \\ \Phi'' \vDash \varphi \\ (\Phi'; e_1 == false) \vDash \varphi \quad (\Phi'; e_1 == false),\Gamma \vdash stmt : \tau \dashv \Phi''',\Gamma'''\end{array}}{\Phi,\Gamma \vdash \text{while } [\exists\varphi](e_1 : bool, e_1)\text{in stmt} : \tau \dashv \Phi''',\Gamma'''}$$

$$\text{T-if}\frac{\begin{array}{c}\Phi,\Gamma \vdash e : bool \dashv \Phi',\Gamma' \\ (\Phi'; e == true),\Gamma' e_1 : \tau_1 \dashv \Phi'',\Gamma'' \\ `(\Phi'; e == false),\Gamma' e_2 : \tau_2 \dashv \Phi''',\Gamma'''\end{array}}{\Phi,\Gamma \vdash \text{if (e : bool) then } e_1 \text{ else } e_2 : \tau_1 \sqcup \tau_2 \dashv \Phi'' \sqcup \Phi''',\Gamma'' \sqcup \Gamma'''}$$

Figure 5.7: Expression Typing Rules cont...

this generated Presburger formula for the regular typestate state.

Rule (T-let- 1 and 2) presents rules for *let* expression for two different cases. If the lhs variable x does not has a *dependent Integer* type, the rule simply updates the value of the static state in a standard way. If the variable has a *dependent Integer* type, the operation on the rhs value v must be applied to the correlated non-programmable type variable n_x. An auxiliary function *Apply* gives an updated value v_{app} for the rhs value v upon the application of this operation. The index environment is updated with a new formula updating n_x. The check, $n_x \in NP(\Phi)$ confirms that n_x is a non-programmable auxiliary variable in the index-environment.

Rule (T-invoke) is the typing rule for *virtual method invocation* expression, $e.m(e_1...e_n)$ of a well formed method. The (T-mdecl) rule defined later, checks for the well formedness of a method (i.e. checks the validity of a signature against its implementation). (T-invoke) typechecks the base expression e in the current context and ensures that its type is either a regular state S_b or a p-typestate p^{x,S_b}. It fetches the declaration of the callee m declared in S_b and extract the signature for the callee. It typechecks that each actual method parameter $e_1...e_n$ satisfies the pre- p-typestate annotation for $\rho_1...\rho_n$ defined in the signature χ. Each of these typechecks, might update the current type-context and the index-environment, thus the rule ensures that each e_i is checked in the environment updated by the check for e_{i-1}. Besides formal parameters $(\rho_1, \rho_2, ...\rho_n)$, the signature also contains typestate annotations for each of the free environment variable $[\rho_{n+1}...\rho_p]$. The rule typechecks each of these free environment variables against their annotated pre- p-typestates, while updating the context during each check. It updates for each actual parameter and the free environment variables, its p-typestate, based on the post annotation in the method signature assuming the correctness of method implementation. Finally, it updates the *index-environment* and the *type-context* according to the post- p-typestate annotations for the method.

Rule (T-update) presents the typing rule for explicit state change expression $e_1 \leftarrow e_2$ *in stmt*, which is used for explicitly updating the values for programmable type variables. The rule typechecks the RHS expression and updates the p-typestate of LHS expression by the p-typestate of RHS (if RHS is an expression and not a p-typestate state) also updating Φ and Γ accordingly.

Rule (T-while), checks for the invariant satisfaction annotated with a **while** expression in an incoming environment, It assumes the invariant after each iteration and at loop exit.

Rule (T-if) is a standard conditional expression rule with an assumption of a join operation over types τ, Φ and Γ. A join over types is defined using subtyping rules (for types related by subtyping relation) or a type $\top$ (for unrelated types). A join relation over Φ is simply a conjunction of formulas in the two environments while it is defined as a union of the co-domain

for a given element in the domain, for the map Γ.

Field, Method and State Well Formedness The method declaration rule (**T-m Decl**) ensures, that each pre- type annotation for method parameters and the free environment variables are well formed. Further, it assumes that each of these variables ρ_i, satisfy their pre-conditions. It then typechecks the body of the method under this assumption, checking that post the execution of the method body, the types of these variables satisfy the post- conditions and the typestate of the body expression satisfy the return type of the method. **T-s Decl** checks the well-formedness of all the types, fields, methods declared in the state. Rule (T-Decl-1) is the well-formedness rule for a field declaration. Rule (T-f Decl-perm) defines the field declaration rule along with an initial permission. This rule and other alias managing rules will be discussed in Section 5.4.5

$$\text{T-m Decl}\ \frac{\begin{array}{c}\Phi,\Gamma \vdash \forall i, \tau_i : \text{Type} \dashv \Phi,\Gamma \\ (\Phi \wedge_{i=1}^{i=n} \phi_i\, \Gamma, \forall i, \rho_i : \tau_i) \vdash \overline{b} : \tau_r \dashv \Phi',\Gamma' \\ (\Phi',\Gamma') \vdash \forall i, \rho_i : \tau_i' \dashv \Phi'',\Gamma''\end{array}}{(\Phi,\Gamma) \vdash M : \forall[\nabla](\forall i, \rho_i : \tau_i) \rightarrow \exists \rho_r : \tau_r\ (\forall i, \rho_i : \tau_i')\{\overline{b}\} \dashv \Phi'',\Gamma''}$$

$$\text{T-f Decl-1}\ \frac{\Phi,\Gamma \vdash \tau : \text{Type} \qquad \tau = Integer\langle n_f \rangle}{\Phi,\Gamma \vdash \tau\ f \dashv \Phi \wedge (n_f == 0), \Gamma; (f : \tau)}$$

$$\text{T-f Decl-perm}\ \frac{\begin{array}{c}\Phi,\Gamma \vdash \tau : \text{Type} \qquad \tau = p^{x,s} \\ S <: s\end{array}}{\Phi,\Gamma \vdash \tau\ f\ =\ new\ S\ \dashv \Phi,\Gamma; (f : (unique, \tau))}$$

$$\text{T-s Decl}\ \frac{\begin{array}{c}\Phi,\Gamma \vdash S \\ \forall f \in \overline{f}.(\Phi,\Gamma) \vdash f \\ \forall m \in \overline{m}.(\Phi,\Gamma) \vdash m \\ \forall \gamma \in \overline{\gamma}.(\Phi,\Gamma) \vdash \gamma\end{array}}{(\Phi,\Gamma) \vdash \text{state S : S' } \{\ \overline{f}; \overline{m}; \overline{\gamma}\} : \tau}$$

5.4.4 Elaboration of Typechecking

The language syntax, the type system and the typing rules define the properties of a "correct" program. In this section, we outline in brief, our typechecking process. The typechecking algorithm/process or the "typechecker" uses the language syntax and types (in form of p-typestate annotations from the programmer), and typing rules to check if a given program with a *main* method correctly satisfies a given p-typestate property. This p-typestate property is not provided as an argument to the typechecker, which is common in typestate analysis approach (see Chapter 4), rather our core typestate oriented programming language allows the programmer to explicitly encode the property in the program using the language's dependent type system. The typechecker takes such an annotated program P and checks if the annotated

typestate property is observed by *state* and *method* declarations and the *main* method. In other words, the typechecker is the process provides a solution to the p-typestate verification problem. Algorithm 2 present a very hhigh-level view of this typechecking algorithm. The algorithm has three major phases, *the constraint generation*, *the invariant calculation* and *constraint solving*. The constraints are generated by assuming all the precedence conditions of an applicable typing rule, and then generating implication of the consequence from the assumption. The details of the loop invariant calculation is described in Section 5.5. Here we elaborate the typechecking algorithm through an example showing how constraints are generated. For clarity and ease of understanding we take our running example program from the Overview section.

Algorithm 2: High level view of the typechecking algorithm

input : p-typestate annotated input program P
output: A proposition (P typechecks)

```
1 generate constraints using type system rules
2 generate loop-invariants using loop invariant calculation technique.
3 result ← solve constraint using a solver like Z3.
4 if result == valid then
5 |   P typechecks
6 else
7 |   P fails to typecheck
8 |   generate violating assignment
9 end
```

5.4.4.1 Generating constraints

In practice, the typing rules inductively generate and transform a set of constraints as Presburger formulas, capturing some p-typestate property. These constraints are then passed to a simple off-the-shelf SMT solver like Z3 [43], which checks for the validity of the constraints to verify that the program satisfies the p-typestate property, or returns a typestate violation error. 688 To explain constraint generation process we present the PStack implementation in DRIP from the Overview section again, and show in Figure 5.8 how the constraint environment Φ ev olves for different expressions and statements in a section of the example program presented in the Overview section. Line 3 and 4 uses the general dependent function introduction rule (T-Pi-I) and updates an empty incoming *index-environment* with a Presburger formula

$$\exists.n_p \in \mathbb{N}, \exists n_r \in \mathbb{N}, \exists s \in Stack\_initialized, \_uninitialized.(n_p \geq n_r \wedge s = Stack)$$

We have dropped the existential quantifiers from the formula in the figure for brevity. Reader should note here that, the state information is also represented as a presburger formula using a state variable s. Statements in lines 5-8 are *field-declarations* and use typing rule for it to add new er auxiliary variables and Presburger formulas defined over them to the index environment. Lines 9-14 similarly uses, typing rule for a *method-declaration*. The rules for a method declaration are gene rated and checked separately in a modular way. The body of a method is checked in an index environment generated assuming the annotated pre- typestate for the method. For example, line 11, shows the such an i ndex environment. Lines 12 and 13 show a p-typestate construction and how the typing rule for typestate construction (T-dep-app) updates the the index-environment. The index-environment post the last statemen t of a method body is checked against the post-typestate to verify the correctness of a method implementation. The constraints are generated in a similar manner for the method declaration for method put in lines 15-24. The constraints for method main, are generated with some incoming global environment (if any).

```
1  state Stack {
2      // a dependent function type with \#p , \#r a formula over it and a possible regular typestate.
3    type PStack :  Pi ( n_p, n_r, n_p ≥ n_r ) -> Stack;
4      {n_p ∈ ℕ, n_r ∈ ℕ, s ∈ Stack_initialized, _uninitialized.(n_p ≥ n_r ∧ s = Stack)}
5     Item[] array = new Item[capacity];
6      var Integer ⟨n_cap⟩ capacity;
7      var Integer ⟨n_top⟩ top; // top index
8      var Integer ⟨n_count⟩ count;
9     {n_p, n_r, s, n_cap, n_count, n_top.(n_p ≥ n_r ∧ n_cap == 0, n_count == 0, n_top == 0, s = Stack)}
10 public Stack()[PStack ( True ) -> _uninitialized >> PStack ( n_p == 0, n_r == 0
       ) -> _initialized  this]{
11     {n_p1, n_r1 ∧ s = _uninitialized ∧ True}
12     return unique PStack (0, 0) -> _initialized st = new Stack();
13     {n_p1 == 0, n_r1 == 0 ∧ s = _initialized ∧ True}
14 }
15 public put (Item x)[ PStack (n_p, n_r) -> _initialized >> PStack
       (n'_p = n_p + 1, n_r) -> \_initialized this, Integer⟨n_top, 0 ≤ n_top < n_capacity⟩ >> Integer⟨n'_top = n_top + 1⟩ top
       ,
16 Integer⟨n_count⟩ >> Integer⟨n'_count = n_count + 1⟩ ]{
17  {n_p1, n_r1, s, n_count, n_top.(n_p ≥ n_r∧, n_count == 0, n_top == 0, s = _initialized)}
18     top = top + 1;
19     array[top] = x;
20     count = count + 1;
21     this <- PStack (n'_p = n_p + 1, n_r) -> _initialized;
22  {n_p1, n_r1, s, n_count, n_top.(n_p1 == 1 ≥ n_r == 0∧, n_count == 1, n_top == 1, s = _initialized)}
23  }
24 }
```

Figure 5.8: Constraints for PStack (Figure 5.2) in DRIP

5.4.5 Handling Aliases

A sound typestate type checker must be aware of all the references to an object in order to precisely capture any possible typestate transition. To track p-typestate changes in the presence

of aliasing we use an earlier developed approach due to Deline et al ([9]) based on a variant of *Linear Types*. With each reference of a type τ in our language, we also assign a label called as *permission*. A permission records whether a given reference to an object is *unique*, in which case it may be used to change the typestate of the system, else if the object is shared amongst more than one references, we assign it an *immutable* permission and do not allow typestate changes to the object's typestate via this reference. A *unique* permission associated with a parameter guarantees to the method that it can access the object only through the current reference. An operation can downgrade (refer Rule P-let-loose) the *unique* permission of a reference to *immutable*. Figure 5.9 presents a set of permission rules for a few of the important expression which mutate references and hence must be tracked for alias confinement. Rule (T-f Decl-perm) adds a *unique* permission to the type of a newly declared field f. Rule (T-let-perm) is the permission assignment rule for a let-expression, the rule update the permission of both the lhs and the rhs to *shared*. Rule (T-update-perm) update the permission of lhs expression by that of the rhs expression, however, the permission of the rhs expression remains unaffected. Rule (T-invoke-perm) confirms that the base object has a unique permission for the method to be invoked.

This simple permission-based alias control and tracking may be made more precise using complex linear types as discussed in *adoption and focus* work([51]). We leave such a study and implementation of advanced alias control type system as a part of future work.

5.5 Calculating Loop Invariants

One of the most important challenges in automatic, inductive type checking of expressive type systems like Liquid types [112], other refinement types, and our p-typestate type system is the requirement to annotate loops and recursive data structures with invariants. These invariants allow to generate modular proofs of correctness for the program in Floyd-Hoare style proofs of program correctness and are fundamental to the termination guarantee of the p-typestate type checking algorithm. Till now we have assumed that loop invariants are being provided by the programmer. Unfortunately, its a daunting task even for experienced programmers to provide such invariants. Moreover, in many cases, the loop invariants provided by the programmer may be too weak and insufficiently precise to verify a given p-typestate property. In fact, there is a trade-off between the ease of guessing the loop invariant and its precision, for example, the easiest and most degenerate loop invariant for a loop may be a tautological formula *true*, but such a formula is a valid loop invariant, but does not add any value to prove a property at the exit of the loop. Thus, it is important to be able to automatically calculate adequate inductive loop invariants for programs whenever possible. To tackle this challenge we present a novel and

$$\text{T-f Decl-perm}\ \frac{\begin{array}{c}\Phi, \Gamma \vdash \tau : \text{Type} \quad \tau = p^{x,s} \\ S <: s\end{array}}{\Phi, \Gamma \vdash \tau\ f\ =\ new\ S\ \dashv \Phi, \Gamma; (f : (unique, \tau))}$$

$$\text{T-let-perm}\ \frac{\begin{array}{c}\Gamma(x) = (a, \tau_x) \\ \Gamma(e) = (a', \tau_e) \\ \Gamma' = \Gamma[x \mapsto (shared, \tau_x), e \mapsto (shared, \tau_x))]\end{array}}{\Phi, \Gamma \vdash \text{let x = e in stmt} \dashv \Phi, \Gamma'}$$

$$\text{T-update-perm}\ \frac{\begin{array}{c}\Gamma(e_1) = (a, \tau_x) \\ \Gamma(e) = (a', \tau_e) \\ \Gamma' = \Gamma[e \mapsto (a, \tau_x)]\end{array}}{\Phi, \Gamma \vdash \text{e} \leftarrow e_1 \text{ in stmt} \dashv \Phi, \Gamma'}$$

$$\text{T-invoke-perm}\ \frac{\begin{array}{c}\Phi, \Gamma \vdash e : (unique, S_b) \mid e : (unique, p^{\{x, S_b\}}) \dashv \Phi_1, \Gamma_1 \\ \Gamma(S_b :: m) = m : \Psi\{\bar{b}\} \\ \Psi = \forall[\triangledown](\rho_1 : \tau_1, ..., \rho_n : \tau_n, \rho_{n+1} : \tau_{n+1}, ..., \rho_p : \tau_p) \\ \rightarrow \exists \rho_r : \tau_r\ (\rho'_1 : \tau'_1, ..., \rho'_n : \tau'_n, \rho'_{n+1} : \tau'_{n+1}, ..., \rho'_p : \tau'_p) \\ \Phi_1, \Gamma_1 \vdash e_1 : \tau_1 \dashv \Phi_2, \Gamma_2 \\ \Phi_2, \Gamma_2 \vdash e_2 : \tau_2 \dashv \Phi_3, \Gamma_3 \\ ... \\ \Phi_n, \Gamma_n \vdash e_n : \tau_n \dashv \Phi_{n+1}, \Gamma_{n+1} \\ \Gamma_{n+1}(\rho_{n+1}) : \tau_{n+1} \\ ... \\ \Gamma_{n+1}(\rho_p) : \tau_p \dashv \Phi'', \Gamma'' \\ \forall\ i \in [1...n].\Gamma''[(e_i) \mapsto \tau'_i] \\ \forall\ j \in [n+1...p].\Gamma''[\rho_j \mapsto \tau'_j]\end{array}}{\Phi, \Gamma \vdash e.m(e_1, e_2, ...e_n) : \tau_r \dashv \Phi'', \Gamma''}$$

Figure 5.9: Typing Rules for Permission Mutations

simple loop invariant calculation approach based on loop acceleration technique for Presburger definable transition systems [15, 54].

5.5.1 Calculating loop-invariants using acceleration for Presburger-Definable Transition Systems

5.5.1.1 Acceleration:

The problem of calculating the reachable set of states (REACH) for an infinite state system is undecidable in general. Thus, model checking infinite state systems requires "symbolic" approach. This involves abstracting a symbolic model of the model checking problem and manipulating it to calculate fixpoints for forward and backward reachability sets. A naive fixpoint calculation for these infinite systems may diverge in general and thus has a low probability of termination. Acceleration [15] is a popular technique which makes the convergence of the fixpoint calculation for such systems more frequent. The technique is analogous to abstract widening operation from the abstract interpretation domain [39].

Definition 5.3 (Acceleration over a path π) Given a transition system $\mathbb{T} = \langle Q, \Sigma, \Psi, \delta \rangle$ and a sequence of action $\pi \in \Sigma^*$. *Acceleration* of π over $\mathbb{T}$ is called π -acceleration and is defined as a relation $Acc_\pi \subseteq (Q \times Q)$ such that (s, s') $\in Acc_\pi$ iff $\exists k \in \mathbb{N}$. such that s $\xrightarrow{\pi^k}$ s', where s $\xrightarrow{\pi^k}$ s' represents a path (s $\xrightarrow{e_1}$ s1 $\xrightarrow{e_2}$ s2... sk-1 $\xrightarrow{e_k}$ s') of k consecutive transitions in the system in δ. We say that s' $\in post_{\mathbb{T}}(\pi^*, s)$ or simply $post^*(s)$, where $post_{\mathbb{T}}(\pi^*, s)$ represents the set of post reachable states by the acceleration of π over $\mathbb{T}$, starting from the initial state s. The definition could be extended to a set of starting states S, by calculating $post^*(s), \forall s \in S$. The acceleration relation Acc_π is called π acceleration or just acceleration when the context is obvious.

5.5.1.2 Computing REACH using acceleration:

The acceleration set defined above can be effectively utilized to calculated REACH for a system. For a given subset of initial states or configurations X of the system, and a language $\mathbb{L} \subseteq \Sigma^*$, we define $post(\mathbb{L}, X) = \{x' \mid \exists x \in X \wedge (x, x') \in Acc_\pi \wedge \exists \pi \in \mathbb{L}\}$. The set $post(\Sigma^*, X)$ of all states/configurations reachable from X(initial set of configurations) is defined as the reachability set REACH of the system.

5.5.1.3 Using REACH for loop invariant calculation:

We model the loop invariant calculation for a program enforcing a p-typestate property as REACH finding problem over the counter system induced by the looping construct in the program. This reduction allows us to use known acceleration based reachable states computation

approaches and tools. We use a Flat acceleration tool FAST [54] in our implementation to calculate REACH for while loops in programs. The reduction is straight forward, Presburger formulas over integer variables of the counter system for the input loop forms the symbolic domain and restrictions on the structure of these counter system guarantees the termination of the FAST tool. FAST calculates a Presburger definable formula representing the REACH set for the input loop with the given initial set of states. We use this formula as a loop invariant in p-typestate type checking to generate a modular proof of correctness of the program.

The approach assumes that the input counter system for the loop is *finite linear* [54] and *flattable* [15]. These restrictions are fundamental to the termination of the approach which uses a semi-algorithm to calculate REACH set. The tool's acceleration algorithm can run on any finite linear system and is a complete procedure for flattable finite linear systems. It provides no termination guarantee for other general class of finite linear but non-flattable systems.

Definition 5.4 (Integer Counter System for Loops) *A counter system C for a loop is a transition system, defined as a tuple $\langle Q, \Sigma, \Psi_{\mathcal{P}}, \delta \rangle$. Where Q is a finite set of states, Σ is a finite set of Presburger definable actions. These actions simulate the p-typestate contracts associated with statements and expressions of the program. $\Psi_{\mathcal{P}}$ is a Presburger definable set over m integer variables $\mathbb{V}^m$ used in the loop. $\Psi_{\mathcal{P}}$ defines guards for transitions given by $\delta : (Q \times \Sigma \times \Psi_{\mathcal{P}}) \mapsto (Q \times \Psi_{\mathcal{P}})$.*

A *state* of C is defined as a tuple $\mathbb{Z}^m$ assigning values to $\mathbb{V}^m$. A state s_i satisfies a Presburger guard $\phi \in \Psi_P$ iff $s_i \vDash \phi$.

Figure 5.10 shows the loop counter system for the while loop in the code fragment for XMLParserSimple defined in Figure 5.16.

5.5.1.4 Complete Approach:

Figure 5.11, shows a block diagram for the invariant calculation approach. The approach requires an integer counter system for the input loop. The counter system can be extracted by analyzing the loop body for pre- and post- p-typestate annotations of expressions and statements and their semantics. In this work we manually construct these counter systems for input loops.

This loop counter system is passed as input to the FAST loop acceleration tool which generates (on termination) the REACH set for the loop as a Presburger formula in the representation of FAST output language Armoise [75]. We built an Armoise to Z3 parser which generates corresponding Z3 formula, and passes it to the type-checker. The type-checker composes the formula with the incoming constraints using the typing rule for while expression. We finally discharge the composed collective constraints to the Z3 solver.

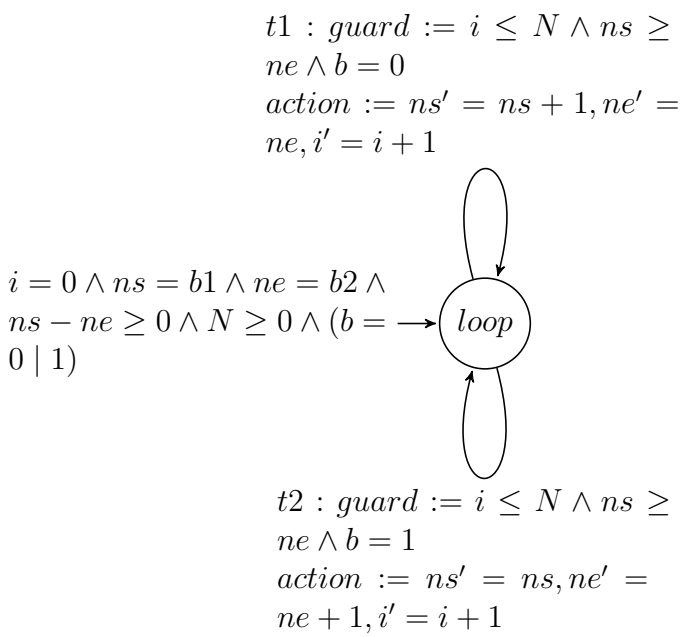


Figure 5.10: Loop counter system for the while loop in Figure 5.16

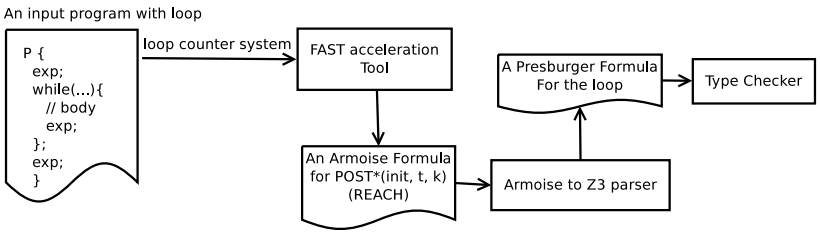


Figure 5.11: Block diagram for loop invariant calculation approach

Example: Consider Figure 5.16 from the Overview section, Figure 5.10 shows the loop counter system for the while loop in the code fragment. The counter system defines the initial set of states, where i is the counter variable for the loop (corresponding to the var numberscanned in the program), b is a boolean variable representing the conditional variable in the loop body. The counter system also defines transitions $t1$ and $t2$, which are Presburger definable actions over index variables. The simplified loop invariant calculated by the approach in this example is $(\forall\ (\mathsf{ns}, \mathsf{ne}, b1, b2, N) \in (nat, nat, nat, nat, nat).\ \mathsf{ns} \leq b1 + N \wedge\ \mathsf{ne} \leq b2 + N \wedge\ \mathsf{ns}\ \text{-}\ \mathsf{ne} \geq 0 \wedge\ \mathsf{ns} \geq b1 \wedge\ \mathsf{ne} \geq b2)$. This loop invariant is indeed adequate to generate an inductive proof of correctness for the main method and prove the main method implementations in Figure 5.3 as correct and incorrect respectively.

```
1  state Broadcast{
2    method sendBroadcast (_ >> _)[_ >> _]{
3      var x, y, z, N;
4      x = N; y = 0; z = 0;
5      while(true){
6          match (*){
7          case (x >= 1) { x = x + y -1; y = z + 1; z
               =0; }
8            case (y >= 0) { y = y -1; z = z + 1; }
9            }default {}
10          };
11     };
12   }
13 }
```

Figure 5.12: **Example: Simple Broadcast** Approach fails to calculate REACH

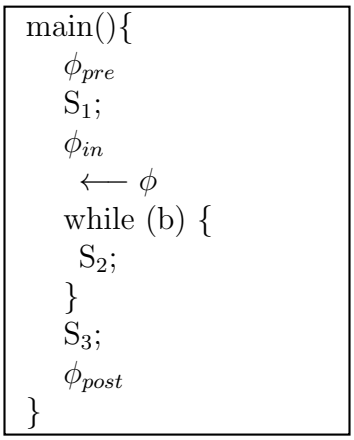


Figure 5.13: Loop invariant ϕ (specified or calculated) for a program with a while loop

It is worth mentioning that loop invariant calculation approach might fail to terminate and return the Presburger definable REACH set for several reasons: (1) The loop counter system is not a *finite linear system*. (2) The counter system is not *flattable*. (3) The approach timed out (for time limit $>$ 3 min). Reasons (1) is ruled out by the assumption of the input loop counter system. The major reason for non-termination therefore is either (2), which although is assumed, but cannot be verified as checking *flattability* of a Counter system is undecidable [54], or, in certain cases the termination is not reached in practical time limits due to (3). Since, checking *flattability* of a counter system is undecidable, we cannot have an algorithm for generating a flattable counter system for an input while-loop. Nevertheless, we can generate a simple counter system for a while loop by using pre- and post- condition annotations for expressions at loop entry, and loop body. Such a counter system still will not guarantee the termination of the reachability set calculation algorithm.

Figure 5.12 presents an example(adapted from [54]) code fragment in our core language with a while loop and two inner conditional expressions for which the approach fails to terminate due to reason (2). More results showing both successful calculations and failures due to reasons (2) or (3) are discussed in section 5.7.

5.5.1.5 Completeness of the loop invariant calculation approach

Given a p-typestate annotated program P with a while loop, we present a claim for the "completeness" of our approach.

Consider Figure 5.13, Each S_i is a block of statements or expressions causing p-typestate constraints changes as depicted in the program. Thus S_1 when executed in a pre- ptypestate with associated constraint ϕ_{pre}, updates the constraints to ϕ_{in}. We abuse the notations here and simply write this transition as $(\phi_{pre}; S_1) \Rightarrow \phi_{in}$. The formula ϕ represents the loop invariant (specified or calculated) for the while loop.

Definition 5.5 (Adequate Inductive Invariant) We say the invariant ϕ is an *adequate inductive invariant* for the given program specification, if following conditions hold-

- $\phi_{in} \Rightarrow \phi$ ("inductive invariant")
- $((\phi \wedge b); S_2) \Rightarrow \phi$ ("inductive invariant")
- $((\phi \wedge \neg b); S_3) \Rightarrow \phi_{post}$ ("adequate")

5.5.1.6 "Completeness" claim:

If our computation of a loop invariant via FAST succeeds, we are guaranteed that it is the "best" possible invariant: if the proof/type-checking does not succeed with this invariant, it cannot succeed with *any* other invariant. This is essentially because this particular loop acceleration technique returns (if it does terminate) the *exact* set of reachable states at the loop head. To substantiate our claim, consider the program structure in Figure 5.13, and suppose FAST returns an invariant ϕ. Suppose further that it is not sufficient (i.e. one of the three conditions above fail). Now by virtue of the fact that ϕ represents the exact set of reachable states at the loop head, it is easy to see that it will satisfy the inductiveness conditions. So it must be the adequacy condition which fails. That is, there exists a state s satisfying ϕ and $\neg b$, such that after doing S_3 we reach a state that does not satisfy ϕ_{post}. But any other proposed invariant ψ must include ϕ, if it satisfies the inductiveness conditions. Hence s must be included in ψ as well, and ψ will similarly fail the adequacy check.

5.6 Analysis

Having discussed the Presburger definable typestates, formal language and the dependent type system implementing the idea, in this section we analyze some important formal properties of our p-typestate typesystem.

5.6.1 Operational semantics

We define a *small-step* operational semantics for the language DRIP described in Section 5.4. The semantics describes the states and transitions associated with the auxiliary integer type variables (both programmable and non-programmable). Such a semantics will facilitate proving the soundness of our p-typestate typesystem. Following is a set of value definitions and maps which we require to define a state of our program

(program variable)	$(PVar)$	:	$x, f, this, \gamma, S, \rho$
(location)	(l)	:	$l_i \mid \textbf{null}$
(type variable)	$(TVar)$	:	$programmable \mid Non-programmable$
(type variable value)	$(TValue)$	:	$\mathbb{N}$
(p-typestate value)	$(PtsValue)$	:	$(\phi, s).p^{\phi,s}$, st $FV(p^{\phi,s}) = \{x_1, x_2....x_n\}$
(value environment)	(μ)	:	$TVar \mapsto TValue$
(program heap)	(Θ)	:	$PVar \hookrightarrow l$
(P-Store-environment)	(Ξ)	:	$l \hookrightarrow PtsValue$

Table 5.3: Value Definitions for States

5.6.1.1 State of a Program

A dynamic state of a program is defined as a tuple (μ, Ξ, Θ). μ is a map from *type variables* to *type variable values*. Θ is a partial map from *program variables* to locations, while Ξ is a partial map which maps these locations to *p-typestate values*. A *p-typestate value* is represented as $((\phi, s).p^{\phi,s}$ st $FV(p^{\phi,s}) = \{x_1, x_2....x_n\})$, making it similar to the p-typestate state type. It represent a value $p^{\phi,s}$ which is a function of ϕ and s with ϕ having type variables $\{ x_1, x_2, ...x_n \}$ possibly occurring free in ϕ.

Each semantic rule is defined as follows:

$$(\mu, \Xi, \Theta)[e] \rightarrow [e'](\mu', \Xi', \Theta')$$

Figure 5.14 presents small-step operational semantics for some of the important DRIP expressions and statements while others are pushed to the Appendix in view of limited space.

$$\text{E-var-2}\frac{\begin{array}{c} x \in domain(\Theta) \\ \Theta(x) = l_x \quad \Xi(l_x) = p^{y,s}\ st\ y = \{y_1, y_2, ..., y_n\} \end{array}}{(\mu, \Xi, \Theta)[x] \rightarrow [\Theta(x)](\mu, \Xi, \Theta)}$$

$$\text{E-de-ref}\frac{\begin{array}{c} \Theta(x) = l_x \qquad l_x \neq \textbf{null} \\ \Xi(l_x) = p^{y,s}\ st\ y = \{y_1, ...y_k, ..y_n\} \\ \Theta(f_j) = l_{fj} \quad \Xi(l_{fj}) = p^{z,s}\ st\ z = \{y_k\} \quad \mu(y_k) = v_{xk} \end{array}}{(\mu, \Xi, \Theta)[x.f_j] \rightarrow [v_{xk}](\mu, \Xi, \Theta)}$$

$$\text{E-let-2}\frac{\begin{array}{c} \Theta(x) = l_x \quad \Xi(l_x) = p^{z,s}\ st\ z = \{z_1, z_2, ..., z_n\} \\ \mu' = \mu[z \mapsto Apply(v, z)] \quad \Theta' = \Theta[l_x \mapsto \Theta(v)] \end{array}}{(\mu, \Xi, \Theta)[let\ x = v\ in\ stmt] \rightarrow [x/v]stmt(\mu', \Xi, \Theta')}$$

$$\text{E-mcall}\frac{\begin{array}{c} \Theta(y) \neq null \qquad \Theta(y) = l_e \\ \Xi(l_e) = p^{x,s}\ st\ \{x_1, x_2, ..., x_n, s = S\} \\ mBody(S, m) = \chi\ \{b\} \quad \chi = \forall[\nabla](\rho_1 : \tau_1, ...\rho_n : \tau_n) \rightarrow \exists \rho_r : \tau_r; (\rho_1 : \tau_1', ..., \rho_n : \tau_n') \\ Check(\chi, \mu, \Xi, \Theta) \end{array}}{(\mu, \Xi, \Theta)[y.m(f_1, f_2, ...f_p)] \rightarrow [b](\mu, \Xi' = \Xi[\Theta(\rho_i) \mapsto \Xi(\Theta(f_i))], \Theta' = \Theta[\rho_i \mapsto \Theta(f_i)])}$$

$$\text{E-update-2}\frac{\begin{array}{c} \Theta(e_2) = l_2 \quad \Xi(l_2) = p^{x_2,s_2}\ st\ x_2 = \{x_{21}, x_{22}, ..., x_{2n}\} \\ \Theta(e_1) = l_1 \end{array}}{(\mu, \Xi, \Theta)[e_1 \leftarrow e_2\ in\ stmt] \rightarrow [stmt](\mu[x_{s1} == s_2], \Xi'[l_1 \mapsto p^{x_2,s_2}], \Theta)}$$

$$\text{E-dep-fun}\frac{\begin{array}{c} FV(x) = \{x_1, x_2, ...x_n\} \\ \mu' = \mu[x_1 == 0, x_2 == 0, ...x_n == 0, z_s == S_{base}] \\ \Theta' = \Theta[x_n ew \mapsto l_x] \qquad \Xi' = \Xi[l_x \mapsto p^{x,s}] \end{array}}{\mu, \Xi, \Theta[type\gamma : \Pi(x : \phi, s : S).p^{x,s}] \rightarrow [skip](\mu', \Xi', \Theta')}$$

$$\text{E-app}\frac{\begin{array}{c} \Xi(\gamma) = \Pi(\phi(x_1, x_2, ..., x_n), s).p^{(\phi,s)} \\ \Xi(e_2) = (\{x_1 = v_1, x_2 = v_2, ...x_n = v_n\}, s = s_2) \end{array}}{(\mu, \Xi, \Theta)[\gamma(e_2)] \rightarrow [p^{\phi_2,s_2}\ st\ \phi_2 = \{v_1, v_2, ...v_n\}](\mu[x_1 = v_1, ...x_n = v_n, s = s_2], \Xi, \Theta)}$$

Figure 5.14: Operational semantics for DRIP

5.6.2 Type Soundness

Definition 5.6 (Safe Typing) *A static typing environment (Φ, Γ) "safely types" a dynamic state behavior of the program (μ, Ξ, Θ) represented as $(\Phi, \Gamma) \vDash (\mu, \Xi, \Theta)$ if the following rule applies:-*

$$P\text{-}cover \frac{\begin{array}{c}\forall x_i \in domain(\mu), \mu(x_i) = n_i, \exists \phi_i = (x_i == n_i) \in \Phi \\ \mu \| \Phi \| \qquad\qquad (\Phi, \Gamma) \vDash (\Xi, \Theta) \\ \forall x_i \in domain(\Theta), \Theta(x_i) \in domain(\Xi) st \\ if \Xi(\Theta(x_i)) = p^{\phi_i, s_i} st FV(\phi_i) = \{y_1, y_2, ..., y_n\} \\ x_i \in domain(\Gamma) \qquad \Gamma(x_i) = p^{(\phi_j, s_2)} \qquad \phi_j \vDash \phi_i \\ s_2 :> s_1 \qquad\qquad \mu[y_1, ...y_n] \| \Phi \|^{|\{y_1 ....y_n\}}\end{array}}{(\Phi, \Gamma) \vDash (\mu, \Xi, \Theta)}$$

$$P\text{-}sat \frac{\forall \phi_i \in \Phi, \forall x_i \in FV(\phi_i), x_i \in domain(\mu), \Phi[x_i \mapsto \mu(x_i)] = true}{\mu \| \Phi \|}$$

$$P\text{-}sat\text{-}restricted \frac{\forall \phi_i \in \Phi, \forall y_i \in \{y_1, ...y_n\} \in domain(\mu), \Phi[y_i \mapsto \mu(y_i)] = true}{\mu[y_1, ...y_n] \| \Phi \|^{|\{y_1 ....y_n\}}}$$

Theorem 5.1 (Preservation) *The language semantics preserves the safe-typing of a well typed program. Formally, we state it as follows- Case : if $e \rightarrow e'$ -*
Iff :

$$(\Phi, \Gamma) \vdash e : \tau \dashv (\Phi_1, \Gamma_1)$$
$$(\Phi, \Gamma) \vDash (\mu, \Xi, \Theta).$$
$$(\mu, \Xi, \Theta)[e] \rightarrow [e'](\mu', \Xi', \Theta')$$

then,

$$(\Phi_1, \Gamma_1) \vdash e' : \tau' \dashv (\Phi', \Gamma') \; and$$
$$(\Phi', \Gamma') \vDash (\mu', \Xi', \Theta')$$

Case if : $e \in value$ and $e = v$

$$(\Phi, \Gamma) \vdash v : \tau$$
$$(\Phi, \Gamma) \vDash (\mu, \Xi, \Theta)$$
$$(\Phi, \Gamma; v : \tau) \vDash (\mu, \Xi, \Theta; x \mapsto v) st \; x \; is \; fresh.$$

Proof: Let us try to prove the above statements for each possibly well typed term e in our language and each possible evaluation rule applicable to the term.

A term e can be -

- Case : e is not a value
  - e can be a statement,
    1. Case[LET] : $(\Phi, \Gamma) \vdash$ t := let x = e_1 in stmt, In this case either of the two evaluation rules (E-let-1 or E-let-2) might apply, lets break on both these cases :

       E-let-1 : (μ, Ξ, Θ) [let x = e_1'] $\rightarrow$ [let x = e_1'] (μ', Ξ', Θ'), where (μ, Ξ, Θ) $[e_1] \rightarrow [e_1']$ (μ', Ξ', Θ') By Induction hypothesis-
       (a) $(\Phi, \Gamma) \vdash e_1' : \tau \dashv (\Phi', \Gamma')$.
       (b) $(\Phi', \Gamma') \vDash (\mu', \Xi', \Theta')$

       Thus by [T-let-1], $(\Phi', \Gamma') \vdash t' : \tau'$ and using I.H. $(\Phi', \Gamma') \vDash (\mu', \Xi', \Theta')$.

       E-let-2 : (μ, Ξ, Θ) [let x = v in stmt] $\rightarrow$ [[x/v]stmt (μ', Ξ, Θ'), where $(\mu' = \mu[z \mapsto Apply(v, z)]$ and $\Theta' = \Theta[l_x \mapsto \Theta(v)]$

       We claim that the $Apply(v)$ function in [T-let-2] along with the update to $\Phi \wedge (n_x == Apply(\text{v}))$ and the static state update due to the checking of $stmtx/v_{app}$, safely covers [using P-cover] the dynamic state change occurring in [E-let-2] via the $Apply(v, z)$ and change to Θ Finally, using this claim, $(\Phi', \Gamma') \vdash t' : \tau'$ and $(\Phi', \Gamma') \vDash (\mu', \Xi, \Theta')$.

    2. Case[UPDATE] : t := e $\leftarrow$ e_1 in stmt, In this case again two evaluation rules are applicable (E-update-1 or E-update-2), let us break on both these cases :

       E-update-1 : t $\rightarrow$ t' := (μ, Ξ, Θ)[e $\leftarrow$ e_1 in stmt] $\rightarrow$ [e $\leftarrow$ e_1' in stmt](μ', Ξ', Θ'). By I.H.
       (a) $(\Phi, \Gamma) \vdash e_1' : \tau \dashv (\Phi', \Gamma')$.
       (b) $(\Phi', \Gamma') \vDash (\mu', \Xi', \Theta')$

       Thus by [T-update-1](refer Appendix), $(\Phi', \Gamma') \vdash t' : \tau'$ and using I.H. (Φ', Γ') $\vDash (\mu', \Xi', \Theta')$.

       E-update-2 : t $\rightarrow$ t' := (μ, Ξ, Θ)[e $\leftarrow$ e_1 in stmt] $\rightarrow$ [stmt](μ, Ξ', Θ). By I.H.

       stmt is well typed.
       (a) By I.H and [T-update], $(\Phi \wedge \phi_1 \wedge (x_i == s_1), \Gamma; e : p^{\phi_1, s_1}) \vDash (\mu', \Xi', \Theta')$.

    3. Case[WHILE] : while - The invariant is sufficient for the proof no update to the dynamic state thus the second part of Preservation statement $(\Phi', \Gamma') \vDash$ (μ', Ξ', Θ') is trivially true(assuming the type and index environment are performing only weak updates on program and type variables, which is true). Hence we just show $(\Phi_1, \Gamma_1) \vdash e' : \tau' \dashv (\Phi', \Gamma')$.

while $[\exists.\phi]$ (e_1) {e}. Again two distinct possible way of reduction of t $\rightarrow$ t'-

* If $e_1 \rightarrow e_1'$, by T-while $e_1' : bool$, and let $(\Phi_1 \wedge (e_1' == true), (\Gamma, e_1' : bool)) \vdash e : (\Phi_2, \tau')$ $\Phi_2 \vDash \exists.\phi$, then t' : τ'.
* If e $\rightarrow$ e', By IH $(\Phi_1 \wedge (e_1 == true), (\Gamma, e_1 : bool)) \vdash e' : (\Phi_2, \tau')$ $\Phi_2 \vDash \exists.\phi$, then t' : τ'.

4. Case[IF] : if - Standard rule, no update to the dynamic state
5. Case[SKIP] : skip

– e can be an expression.

1. Case [CALL] : t = e.m$(e_1, e_2, ...e_p)$, The proof uses the operational semantics for [E-mcall], [E-comp](refer Appendix) and the typing judgments for [T-mdecl], [T-comp] and [T-invoke]. By I.H. the preservation holds for each $e_1, ..., e_n$ and the base expression e. Using [E-mcall], we have t $\rightarrow$ t' $(\mu, \Xi, \Theta)[y.m(f_1, f_2, ...f_p)] \rightarrow [b](\mu, \Xi' = \Xi[\Theta(\rho_i) \mapsto \Xi(\Theta(f_i))], \Theta' = \Theta[\rho_i \mapsto \Theta(f_i)])$. The [T-invoke] rule explicitly updates the type environment Γ and index environment Φ so that each actual parameter satisfy the pre- contract required for the method call. Thus the updated environment safely covers the semantics of the [E-mcall].
   Using the semantics for [E-comp] and [T-mdecl] and [T-comp], we prove the preservation for after the call expression.
2. Case [APP] : t $\rightarrow$ t' := $(\mu, \Xi, \Theta)[\gamma(e_2)] \rightarrow [p^{\phi_2, s_2}\ st\ \phi_2 = \{v_1, v_2, ...v_n\}](\mu[x_1 = v_1, ...x_n = v_n, s = s_2], \Xi, \Theta)$. The [T-app] rule, update the Index environment Φ with $\Phi' = \Phi \wedge pr_1(e_2) \wedge \Omega(pr_2(e_2))$, using [P-cover], we get $(\Phi', \Gamma) \vDash (\mu[x_1 = v_1, ...x_n = v_n, s = s_2], \Xi, \Theta)$.
3. Case [DEREF] : The [E-de-ref] does not updates the dynamic state, thus the preservation is triviall

- Case : e is a value. If e is a value then there is no e $\rightarrow$ e' and the preservation is vacuously true.

□

Theorem 5.2 (Progress) *The language small step operational semantics guarantee the reduction of a well typed term to a program value. We state this formally as follows-*

Iff :

$$(\Phi, \Gamma) \vdash e : \tau \dashv (\Phi', \Gamma') \; then$$
$$either \; e \in value$$
$$or, \exists e', \; s.t.$$
$$(\mu, \Xi, \Theta)[e] \rightarrow [e'](\mu', \Xi', \Theta')$$

Proof: The idea for the proof of progress is similar to that for the preservation, we prove that the progress statement holds for each possibly well typed term in the language and each reduction of the term.

- Case : e is not a value.
  - e can be a statement,
    1. Case[LET] : t := let x = e_1 in stmt.
       By I.H. the Progress statement holds for e_1, thus e_1 is either a value or there exists an e_1', for both these cases, the induction step for t is given by rule [E-let-1] and [E-let-2] respectively.
       * E-let-1, t' = let x = e_1' in stmt.
       * E-let-2, t' = [x/Apply(e_1)]stmt.
    2. Case[UPDATE] : t := e $\leftarrow$ e_1 in stmt, Again by I.H. over e_1, e_1 is either a value v or there exist a term e_1', such that $e_1 \rightarrow e_1'$. Two of these cases are defined by evaluation rules [E-update-1] and [E-update-2].
       * E-update-1, t $\rightarrow$ t' := (μ, Ξ, Θ)[e $\leftarrow$ e_1 in stmt] $\rightarrow$ [e $\leftarrow$ e_1' in stmt](μ', Ξ', Θ').
       * E-update-2, t $\rightarrow$ t' := $(\mu, \Xi, \Theta)[e_1 \leftarrow e_2 \; in \;$ stmt] $\rightarrow$ [stmt]$(\mu, \Xi'[l_1 \mapsto p^{x_2, s_2}], \Theta)$.
    3. Case[WHILE] : Standard [E-while] evaluation rule.
    4. Case[IF] : Standard reduction
    5. Case[SKIP] : a value,
  - e can be an expression, we discuss those expressions which are not a value.
    1. Case[DE-REF] : t := e.f, By I.H. either e is a value or there exists e', following are the two cases-
       * Case e $\rightarrow$ e', t $\rightarrow$ t', such that t' := e'.f.
       * Case e is a value, Rule [E-de-ref], defines the reduction for t $\rightarrow$ t'.

- Case[CALL] : t := e.m($e_1, e_2, ...e_p$), By I.H., each of the e and $e_1, ...e_p$, either is a value v_i or there exists a reduction $e_i \rightarrow e_i'$, giving a reduction t $\rightarrow$ t', where t' := e'.m($e_1', e_2', ...e_n'$). The other case where each of e and e_i is a value is reduced by the evaluation rule [E-mcall].
- Case[APP] : t := $\gamma(e_1)$, By I.H. either e_1 is a value (which is a pair by inverse typing rules) or there exists e_1', such that $e_1 \rightarrow e_1'$.
- Case[PAIR] : A pair is a value.
- Case[CONSTANT] : A constant is value.

• Case : e is a value. This is vacuously true.

□

Theorem 5.3 (Soundness) *The typestate system presented in section 5.4 is sound. Formally, if a term* ***t*** *is a well typed term in our typestate system, then it will never be a stuck term, more specifically the term is guaranteed to satisfy all the pre and post p-typestate contracts associated with instructions accessing the term (protocol implementation soundness), further if a usage (main method) might violate any contract for any term in the program, the static type system will Ill type the implementation (protocol usage soundness).*

Proof: By Theorem 5.1 and 5.2 □

5.6.3 Annotation Overheads

One of the major limitations of a verified design-by-contract system and language like DRIP and earlier such systems is the burden of contract annotation on the programmer. The problem is common to the languages with richer types, like *refinement types* [112], however since these types are mostly defined for functional languages with no side-effects, inferring some portion of type and refinement annotations is relatively easier in their case as compared to DRIP. We plan to apply some of these techniques which are used for learning refinement types([129]) to DRIP in our future work.

Another important challenge in automatic, inductive type checking of expressive type systems like Liquid types [112], other refinement types, and our p-typestate type system is the requirement to annotate loops and recursive data structures with invariants. These invariants are fundamental to the termination guarantee of the p-typestate type checking algorithm. Unfortunately, providing these invariants is hard, and in many cases the loop invariants provided by the programmer may be too weak and insufficiently precise to verify a given p-typestate

property. To tackle this challenge we developed a novel loop invariant calculation approach based on loop acceleration technique for Presburger definable transition systems [15, 54]. The details of the approach are beyond the scope of this work and cannot be presented in the view of limited space. Interested readers should refer to the Appendix, which provides a high level view of the approach. We also present the loop-invariant calculation results in section 5.7.

5.7 Results

In this section, we present a small set of useful programs with important non-regular program properties which cannot be modelled using regular typestate but have been statically verified using p-typestates.

5.7.1 Implementation

The parser for the core language concrete syntax has been implemented using *JavaCC* [69] parser generator [1]. The prototype p-typestate type checker has been implemented in Java and contains three major components. (i) A Presburger constraint generator which generates constraints based on p-typestate typing rules. We showed earlier via examples how these constraints are constructed using typing rules and further discussed the approach more formally in the Analysis section 5.6. (ii) A loop invariant calculator, described earlier which uses the FAST tool. (iii) A Presburger constraint solver. The constraint generator has been written in Java. The loop invariant calculator requires an Armoise (internal formula representation language of FAST) to Z3 parser and this parser has been written in Java. We use Z3 solver for checking the validity the generated constraints.

5.7.2 Results

Tables 5.4, 5.5, present a set of statically verified p-typestate properties and invariants which we have implemented in our language. Each of these implementations has a set of states, with field, method and type declarations, modelling a p-typestate property automaton for the property and a *main* method with a sequence of language statements and expressions. The typechecker verifies each of the states, methods, fields, and type declarations, thereby verifying *protocol implementation correctness*. Further, it uses these verified declarations to inductively verify, that the program in the *main* method satisfies the p-typestate property automaton, thereby verifying a *protocol usage correctness* which we also call *p-typestate property verification* (refer Section 5.3). The table gives property names, their short description (formal/informal) and reports a calculated loop invariant if applicable, or a timeout using the loop invariant

[1] Available at - https://github.com/ashishtheaegis/p-typestate-parser.git

Property Name	Property Description	LI*
Producer-Consumer	Producer(P), Consumer(C), Shared-Channel(sc) items produced (np), items consumed (nc) S(sc) = {open, close}, sc.produce() iff open, sc.consume() requires iff open $\phi = \forall op \in \{open, close, produce, consume\}^*$. np $\geq$ nc	$\forall np, nc \in \mathbb{N}, \exists k1, k2, N \in \mathbb{N}.(np - nc) \geq (k2 - k1) \wedge k1, k2 \in [0, N]$
SizedArray and SizedList	Array sized N (A(N)), i_{th} index access (A[i]) $\phi_1 = \forall A[i].\ i \leq$ N ϕ_2 = A.add() iff N $\leq k \in \mathbb{N}$	$\forall i \in \mathbb{N}.\exists N \in \mathbb{N}.(0 \leq i \leq N)$
Binary Search	Binary search implementation using SizedArray. No runtime array bounds checks needed	$\forall i \in \mathbb{N}.\exists N \in \mathbb{N}.(0 \leq i \leq N/2) \vee (N/2 + 1 \leq i \leq N)$
List Reversal and Append	List Sized N1 (L1(N1)), List Sized N2 (L2(N2)) L1 = $[x_1, x_2..., x_{N1}]$, L2 = $[y_1, y_2..., y_{N2}]$ L = $[x_1, x_2..., x_{N1}, y_{N2}, y_{N2-1}..., y_1]$ ϕ = size(L) = N1 + N2	$\forall N1, N2 \in \mathbb{N}.\exists size \in \mathbb{N}.(size = N1 + N2)$
Banking Problem	Account (Acc), Amount Withdrawn (W), Amount Deposited (D) S(Acc) = {active, inactive}, Acc.withdraw() iff active, Acc.deposit() iff active $\phi = \forall$ transactions. D $\geq$ W	$\forall D, W \in \mathbb{N}, \exists k1, k2, N \in \mathbb{N}.(D - W) \geq (k2 - k1) \wedge k1, k2 \in [0, N]$
Train Running Protocol	Train speed control protocol [15]	S
StackModel [Figure 5.21]	Simulating a Stack with pushm, pop, top, empty and acceptance using p-typestates	N/A

Table 5.4: Statically Verified p-typestate Properties. *- LI = (Loop Invariant) S = Success, F = Failure, T/O = Timed Out, N/A = No loops

XMLParser	XMLParser (P), open tag $\langle el_i \rangle$ close tag $\langle /el_i \rangle$ $\phi = \forall \sigma \in \{\langle el_i \rangle, \langle /el_i \rangle\}^*.\ \sigma \in \text{Dyck}_m$	N/A
XMLParser-Simple	XMLParser (P), open tag $\langle el \rangle$ close tag $\langle /el \rangle$ $\phi = \forall \sigma \in \{\langle el \rangle, \langle /el \rangle\}^*.\ \sigma \in \text{Dyck}_2$	$\forall ns, ne \in \mathbb{N}, \exists k1, k2, N \in \mathbb{N}.(ns - ne) \geq (k2 - k1) \wedge k1, k2 \in [0, N]$ where $ns =$ number of $\langle el \rangle$ and $ne =$ number of $\langle /el \rangle$
CFL-Parser ($a^n b^n$ -Parser)	Given a graph G, with nodes $\{ n_1, n_2, ..., n_N \}$ and each edge labeled $l \in \{a, b\}$, let $\pi(p) =$ string generated by labels of a path p $\phi_1 = \forall p \in \{a, b\}^*$ s.t. $\exists$ a path p b/w n_i and n_j, s.t. $z = \pi(p) \Rightarrow$ $\phi_1 = z \in a^n b^n$ $\phi_2 = \forall z'$, s.t. $z = z'x$, $z' \in a^j b^k$, where $j \geq k$	$\forall n, j, k \in \mathbb{N}.(j \leq n \wedge k \leq n \wedge 0 \leq j, k \wedge j \geq k$
IVP checking	Given an interprocedural control flow graph G, with edges labeled with method calls (c_i) and returns (r_i). $\phi_1 = \forall \sigma \in \{el_i, /el_i\}^*.\ \sigma \in \text{Dyck}_m$ $\phi_2 = \forall \sigma'$ s.t. $\sigma = \sigma'.x$ $\text{D}(\sigma') \geq 0$	N/A
Broadcast	Simple Broadcast example Figure 5.12	F
Synchronized Inc/Dec	Integer counter shared between two processes which increments and decrements the counters.	TO ($>$ 3 mins.)

Table 5.5: Statically Verified p-typestate Properties. *- LI = (Loop Invariant) S = Success, F = Failure, T/O = Timed Out, N/A = No loops

calculation approach for the property. Some of the property names are annotated with the figure representing its implementation. Each of these examples is approximately 100-200 lines of code in our concrete language syntax and took a few seconds for Presburger constraints generation by the type-checker and solving using Z3 on an Intel Xeon, 8 core CPU@2GHz with 3 GB of memory. We make each of these properties and their implementation in our language, available online at [8]. We briefly explain each of these here:

XMLParserSimple XMLParserSimple is a variant of XMLParser in Java, over a binary set of XML elements, $\Sigma = \{ \langle el \rangle, \langle /el \rangle \}$. A string $z \in \Sigma^*$ is *well-formed* iff, $\forall z = xy$, $D(x) \geq 0$ and $D(z) = 0$, where $D(p) = (\#\ \langle el \rangle \text{ in } p \text{ - } \#\ \langle /el \rangle \text{ in } p)$ is called the *Dyck distance* of a string p. The problem is beyond the expressiveness of regular typestates as argued earlier in the Overview section.

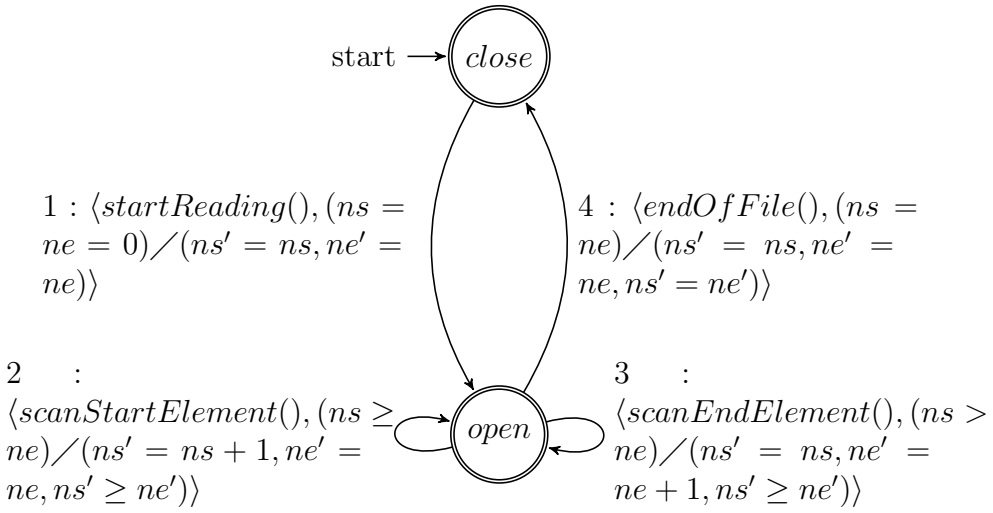


Figure 5.15: A p-typestate Property Automaton for XMLParserSimple

The property is formally defined by a *p-typestate property automaton* in Figure 5.15. Counters ns and ne count the number of start and end elements scanned. The boolean field isOpen stores the *open* or *close* states of the parser. Method startReading starts reading the input document changing the state of the parser from *close* to *open* and is valid only in the *close* state of the parser. Methods scanStartElement and scanEndElement scan a start and end element respectively and are valid in the *open* state of the parser iff the substring z scanned thus far has *Dyck dictance* greater than or equal to zero, i.e. ns $\geq$ ne. Finally, method endOfFile checks the terminal condition for well-formedness of z, i.e. ns == ne.

Figure 5.16 presents an implementation of a simple variant of an XML parser in our language.

```
1  state XMLParserSimple case of Parser{
2
3          type SafeSimpleParser : Pi (ns, ne, ns ≥ ne) → Parser;
4          List<Element> elementsScanned = new ArrayList<Element>();
5          Boolean isReading;
6          method unique Element startReading()[unique SafeSimpleParser (n, m) → Close ≫ unique
               SafeSimpleParser (n, m) → Open]{
7                  this.isReading = true;
8                  this <- unique SafeSimpleParser(n, m) -> Open;
9          }
10         method unique Element scanStartElement(unique StartElement selement)[unique SafeSimpleParser
               (n, m, n ≥ m) → Open ≫ unique SafeSimpleParser (n'=n+1, m'=m, n' ≥ m') → Open]{
11                 elementsScanned.add(selement);
12                 this <- unique SafeSimpleParser(n+1, m) -> Open;
13         }
14         method unique Element scanEndElement(unique EndElement eelement)[unique SafeSimpleParser (n,
               m, n > m) → Open ≫ unique SafeSimpleParser (n'=n, m'=m+1, n' ≥ m') → Open]{
15                 elementsScanned.add(eelement);
16                 this <- unique SafeSimpleParser (n, m+1) -> Open;
17         }
18         method boolean endOfFile()[unique SafeSimpleParser (n, m, n == m) → Open ≫ unique
               SafeSimpleParser (m'= m, n'= n, n' == m' ) → Close]{
19                 this.isReading = false;
20                 this <- unique SafeSimpleParsr(n , m, n == m) -> Close;
21         }
22 }
23
24 method void main(){
25         var unique SafeSimpleParser (0, 0) → Close sP =
26           new XMLParserSimple{elementsScanned = new ArrayList<Element>(); isReading=flase;};
27           List<Element> storedElements = new ArrayList<Element>();
28
29         sP.startReading();
30         sP.scanStartElement(new StartElement());
31         sP.scanStartElement(new StartElement());
32         sP.scanStartElement(new StartElement());
33         sP.scanStartElement(new StartElement());
34
35         sP.scanEndElement(new EndElement());
36         var numberScanned = 1;
37         val N = storedElements.size();
38         while(numberScanned <= 3){
39         Element el =
40         storedElements.remove();
41         match(el){
42         case (StartElement){
43         scanStartElement(el);
44         }
45         case (EndElement){
46         scanEndElement(el);
47         }
48       default {}
49      numberScanned++;
50     };
51   };
52   sP.scanStartElement(new StartElement());
53 }
```

Figure 5.16: An XMLParserSimple implementation and its usage

Producer-Consumer Given a shared Channel object sc and two processes, a Producer named P, and a Consumer named C, the Producer-Consumer property checks that a Channel sc remains in a consistent for all possible sequence of operations performed by P and C over sc. The operations performed by these processes may be, opening a Channel (open), closing a Channel (close), producing an *item* over the Channel by P (produce) and consuming a produced item from the Channel by C (consume). The channel may either be an OpenChannel or a ClosedChannel. Let state(Channel), be an auxiliary method returning the regular state of the Channel and let np and nc, represent the number of items produced and consumed respectively over sc at any instant. A Channel is consistent iff :

- An item is produced or consumed only over an *OpenChannel*.
- For all sequence of operations over sc, (np $\geq$ nc).

Figure 5.17, shows the above defined property in a concise format using a p-typesate property automaton.

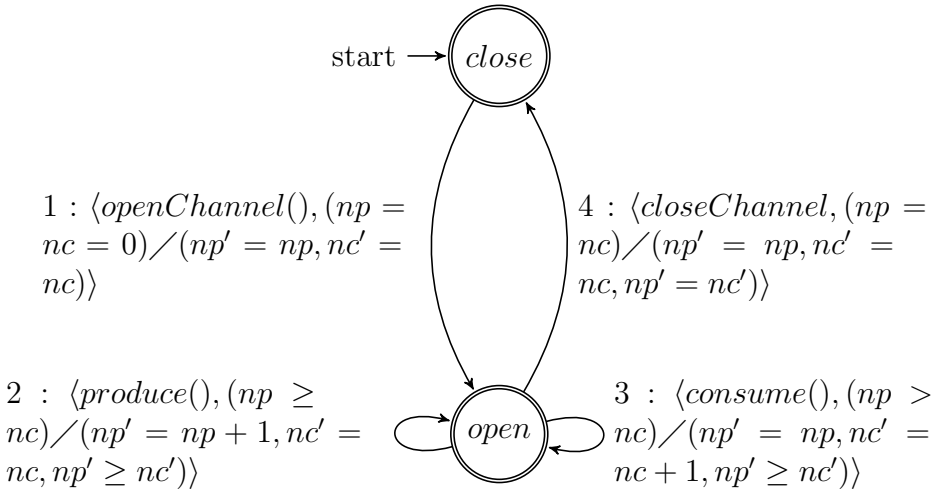


Figure 5.17: A p-typestate Property Automaton for Producer-Consumer program

Binary Search Given an Array, A[0...n] or a List, L[0...n] of n elements, a classic *binary search* algorithm searches for a given input element in the array. It recursively searches for the element in two sub-arrays A[0..n/2] and A[n/2 +1 ...n]. Each of these searches is prone to array bound violations, meaning trying to read beyond an array boundary. Classically, these bound violations are checked using runtime array bound checks. We present an implementation of *binary search*, using SizedArray and SizedList, which statically guarantees that there are no possible array bound violations. This obviates the need for costly and error-prone runtime array bounds checks.

Figure 5.18, shows the above defined property in a concise format using a p-typesate property automaton.

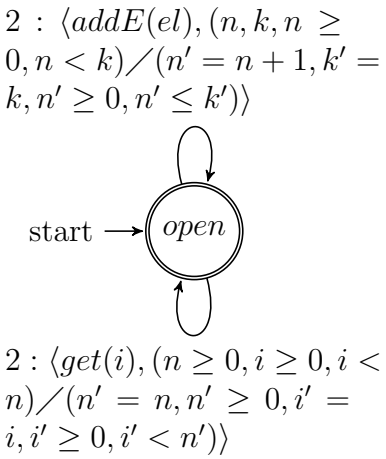


Figure 5.18: A p-typestate Property Automaton for SizedArray access (Used in Binary Search)

List Reversal and Append Given two lists L1 and L2, the list reversal and append operation, reverse the second list L2 and appends it to the first list L1. We present an implementation of this operation in our language to statically verify, that the size of the new list is sum of the sizes of the two sublists. The property is important since it requires to statically tracking the size of lists at the same time, this example presents an expressive limitation of our p-typestate system. We can verify a property, that the new list has L2 in reversed order. Such a property requires a type system which can keep track of relationships between elements which is not possible using a Presburger definable p-typestate. We discuss this limitation and a possible solution in our future work section.

Banking Problem The banking problem presents a problem similar to the *producer-consumer* property as it requires counting and comparing number of operations, named Deposit (D) and Withdraw (W), in a given sequence of operations. The state of a bank Account named Acc may be either active or inactive. An account Acc is consistent iff:

- All the deposits and withdrawals happen over an active account.
- For all sequence of operations, number of Deposit (D) $\geq$ number of Withdraws (W).

This example application shows the ubiquitous nature of the problem as it appears in different domains. We present an implementation of the problem in our language which statically guarantees the consistent state of an Acc.

Train Speed Control Protocol The train speed control algorithm controls the speed of the train and guarantees the collision-free running of the trains. A train could be in one of the four states *viz. ontime, braking , late or stopped.* Thus a safety property for such a control system could be defined as - *"the train is never late (or early) by more than 20 seconds"*. The speed control system is regulated via counters keeping track of number of beacons b passed on the rails and global clock tick s, besides this there is another counter which starts in the braking state and counts the ticks during breaking state d. Each state is defined as - The train is ontime iff $s-9 < b < s+9$, its late iff $b \in [s-9, s-1]$, its early iff $b \geq s+9$ finally, when $b = s+1$, the train is on time again.

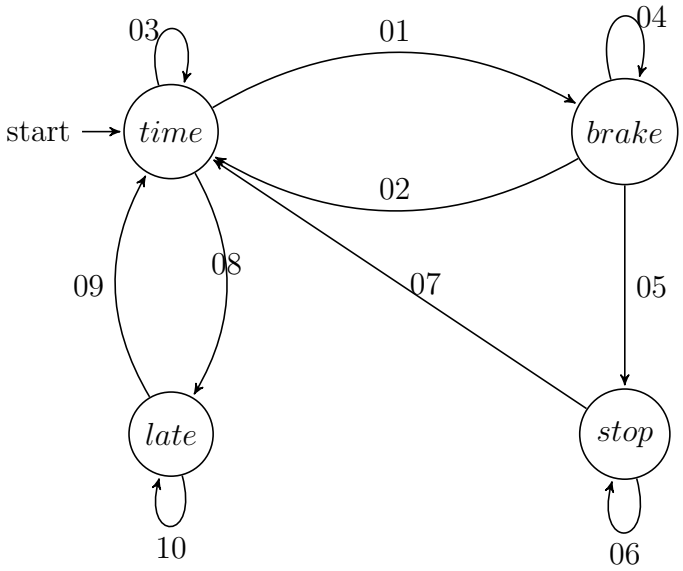
One property of interest to avoid collisions is- $\forall time, \mid b - s \mid \leq 20$, which could not be enforced using regular typestate. We present a counter machine for the train speed control protocol in Figure 5.19.

CFL-Parser $(a^n b^n)$ We implement a Context Free Language parser over a graph in our dependently typed language. Given a graph G, with nodes $\{ n_1, n_2, ..., n_N \}$ and each edge labeled $l \in \{a, b\}$, let $\pi(p)$ = string generated by labels of a path. The parser checks that for any path p, the string $\pi(p)$ generated by the path belongs to the Context-free language $a^n b^n$. We use p-typestates to verify such a property by verifying following two properties over $\pi(p)$ and each substring of it:

- $\phi_1 = \forall p \in \{a, b\}^*$ s.t. $\exists$ a path p b/w n_i and n_j, s.t. $z = \pi(p) \Rightarrow \phi_1 = z \in a^n b^n$
- $\phi_2 = \forall z'$, s.t. $z = z'x$, $z' \in a^j b^k$, where $j \geq k$ $\forall n, j, k \in \mathbb{N}.(j \leq n \wedge k \leq n \wedge 0 \leq j, k \wedge j \geq k$

Interprocedural Valid Path Checking Given an interprocedural control flow graph G, we abstract over method call and return edges in G. A path over such an abstract graph is interprocedurally valid path iff, the sequence generated by the call and return edges along the path belong to the language of matching parenthesis. We implement a path verifier in our p-typestate oriented language to statically verify iff a path is interprocedurally valid. We can see that the problem is similar to the XMLParser property, thus the limitations discussed apply in this case as well.

Broadcast The problem involves two processes, and three shared variable x, y, z between them, and depending on the value of *xandy* either the first or the second process updates the shared variables and broadcast the updated value. The example is crucial as it shows, limitation of our loop invariant calculation approach. The limitation involves inability of FAST approach to calculate the REACH set for the system. A simple code fragment showing the model of the system is shown in Figure 5.12.



01	$\langle (b = s + 9)/(b' = b + 1 \wedge d' = 0) \rangle$
02	$\langle (b = s + 1)/(s' = s + 1 \wedge d' = 0) \rangle$
03	$\langle \{(b < s + 9)/(b' = b + 1)\} \vee \{(b > s - 9)/(s' = s + 1)\} \rangle$
04	$\langle \{(d < 9)/(d' = d + 1) \wedge (b' = b + 1)\} \vee \{(b > s + 1)/(s' = s + 1)\} \rangle$
05	$\langle (d = 9)/(b' = b + 1) \rangle$
06	$\langle (b > s + 1)/(s' = s + 1) \rangle$
07	$\langle (b = s + 1)/(s' = s + 1 \wedge d' = 0) \rangle$
08	$\langle (b = s - 9)/(s' = s + 1) \rangle$
09	$\langle (b = s - 1)/(b' = b + 1) \rangle$
10	$\langle (b < s - 1)/(b' = b + 1) \rangle$

Figure 5.19: A p-typestate property automaton for train speed control system, property $\mid b - s \mid \leq 20$

Synchronized Increment and Decrement The problem again involves two processes sharing an integer counter and incrementing and decrementing the counter. The example is taken from the original work on FAST [54] and again shows the limitation of the loop invariant calculation approach. The tool times out while calculating the REACH set with a timeout threshold of 3 minutes.

SizedList Figure 5.20 presents a sized version of a List with a p-typestate SizedListTy, capturing the size of a List. Each method of the SizedList state is annotated with a pre- and post- p-typestates. For example, method append takes an input list of a p-typestate (unique SizedListTy(m) → List), representing a list of Size m and requires the size of base List object to be n and returns a List of size m+n. The type checking algorithm statically verifies these pre- and post p-typestate annotations. We have implemented sized versions of arrays and lists as a library where sizes are checked and enforced using the p-typestate type system.

```
1  state SizedList case of List{
2          type SizedListTy : Pi (n) -> List;
3          var SizedCons head;
4          var SizedList tail;
5          method void prepend(elem)[unique SizeListTy(n) -> List >> unique SizedListTy(n+1) -> List]{
6                  this.head = new SizedCons {var value = elem; var SizedCons next = this.head;};
7                  this <- SizedListTy(n+1) -> List;
8          }
9          method void add(elem)[unique SizedListTy(n) -> List >> unique SizedListTy(n+1) -> List]{
10                 this.tail = new SizedList {var head = new elem; var tail = new plaid.lang.NIL;};
11                 this <- SizedListTy(n+1) -> List;
12         }
13         method void append(unique SizedListTy(m) -> List list)[unique SizedListTy(n) -> List >>
               unique SizedListTy(n+m) -> List]{
14                 match (list.tail){
15                         case Nil{
16                         this.tail = this.tail.add(list.head);
17                         this <- SizedList(n+1) -> List; }
18                         case Cons{
19                         this.tail = this.tail.append(new Cons {var value = list.head.value; var next
                               = list.tail;});
20                         this <- SizedList(n+m) -> List; }
21                         default{java.lang.System.out.println("bad");}
22                 };
23         }
24         method reverse()[unique SizedListTy (n) -> List >> unique SizedListTy (n) -> List]{
25                 match (this){
26                         case Nil{ this; }
27                         case Cons{
28                                 new Cons{var value = this.tail.reverse(); var next = this.tail;};
29                         }
30                         default{this;}
31                 };
32         }
33 }
```

Figure 5.20: Statically checked SizedList using p-typestate

StackModel Figure 5.21 presents an intricate example. Here we simulate a stack using counters. The StackModelType represents a p-typestate, modeling a stack, with (c1, c2) representing two binary strings, such that, the least significant bits of c1 and c2 defines the top element of the stack. For example, if c1 = "0b00" and c2 = "0b01", then "01" is the stack top. Each stack element(StackElementWithId) has an Id, given by a binary string b1b2. Thus at any time, strings c1, c2 define the contents of the stack. Pushing an element with Id (b1b2) is achieved by updating c1 and c2 to c1' and c2' respectively by multiplying each by 2 and adding b1 and b2 respectively. To pop an element, c1, c2 are divided by 2. Following the definition of StackModel, the main program shows a code sequence creating call and return elements (c_i and r_i), and a sequence of push and pop operations. Our typechecker successfully possible violation of the p-typestate property by the given code sequence. It is worth noting that, our prototype typechecker for the language at present does not has support for these multiplication and division operations over binary strings, thus some part of this example is verified manually. We plan to update the implementation to capture these binary string operations. This may require a sound formulation for capturing of binary arithmetic using Presburger arithmetic formulas.

We use this StackModel to model and verify various important context-free properties like XMLParser, CFL-Parser for a context-free language $a^n b^n$, etc.

XMLParser Given an alphabet set $\Sigma = \{ \langle e_1\rangle\langle/e_1\rangle, \langle e_2\rangle, \langle/e_2\rangle, ...\langle e_m\rangle, \langle/e_m\rangle \}$ of m types of opening and closing XML tags, let $\sigma \in \Sigma^*$. We use the StackModel to implement a general XML parser to verify that, a string σ belongs to the general Dyck_m language. Note here, that we assume that each of these opening and closing tags are read sequentially in the program, i.e. the program reads these symbols on after the another without loops. This restriction can be attributed to the limitation of our loop-invariant calculation approach which cannot handle the loop automata for such a general case.

5.8 Immediate Future Work

There are several directions for an immediate extensions of the work discussed in this chapter. Extension of the p-typestates typesystem with gradual typing will aid in increasing the precision of the verification and will also aid in reducing the burden of annotation on the programmer. A modular p-typestate type system on the lines of Deline et. al. [82] will be really useful. Another interesting direction of future work will be to extend the p-typestate and core language with richer features like polymorphic types, concurrency, etc. This extension will lead to further investigation on the connections between *Session types* [120], normal typestates and p-typestate. A complete formal characterization and applications of the loop invariant calculation approach

```
1          state StackModel{
2          var Stack k1 = new Stack; // regular stack
3          var Stack k2 = new Stack;
4          type StackModelType : Pi (c1, c2) -> Stack;
5          type StackElementWithId : Pi (b1, b2) -> BoundedInteger;
6          // Apply binary pop on both the objects
7          method void push(unique StackElementWithId (b1, b2) -> BoundedInteger element) [unique
               StackModelType (c1, c2) -> Stack >> unique StackModelType (c1', c2' , t1 = c1 * 2, t2 =
               b1 * 1, c1' = t1 + t2 , t3 = c2 * 2, t4 = b2 * 1, c2' = t3 + t4) -> Stack]
8          {
9          this.k1.push(b1);
10         this.k2.push(b2);
11         this <- StackModelType (c1', c2', c1' = (c1 * 2) + (b1 * 1 ), c2' = (c2 * 2) + (b2 * 1)) ->
               Stack;
12
13         }
14         method void pop(unique StackElementWithId (b1, b2) -> BoundedInteger element) [unique
               StackModelType (c1, c2, b1 == c1, b2 == c2) -> Stack >> unique StackModelType (c1', c2',
                c1' = c1 / 2, c2' = c2'/2) -> Stack]{
15         this.k1.pop();
16         this.k2.pop();
17         this <- StackModelType (c1', c2', c1' = c1/2 , c2' = c2/2 ) -> Stack;
18         }
19 }
20       method void main(){
21         var unique StackElementWithId(0b0 , 0b1) -> BoundedIneger c1 = new BoundedInteger;
22         var unique StackElementWithId(0b1 , 0b0) -> BoundedIneger c21 = new BoundedInteger;
23         var unique StackElementWithId(0b1 , 0b1) -> BoundedIneger c31 = new BoundedInteger;
24         var unique StackElementWithId(0b1 , 0b1) -> BoundedIneger c32 = new BoundedInteger;
25         var unique StackElementWithId(0b1 , 0b0) -> BoundedIneger c22 = new BoundedInteger;
26         var unique StackElementWithId(0b1 , 0b0) -> BoundedIneger r21 = new BoundedInteger;
27         var unique StackElementWithId(0b1 , 0b1) -> BoundedIneger r31 = new BoundedInteger;
28         var unique StackElementWithId(0b1 , 0b1) -> BoundedIneger r32 = new BoundedInteger;
29         var unique StackElementWithId(0b1 , 0b0) -> BoundedIneger r22 = new BoundedInteger;
30         var unique StackElementWithId(0b0 , 0b1) -> BoundedIneger r1 = new BoundedInteger;
31
32         var unique StackModelType (0b0, 0b0) -> Stack sm = new Stack;
33         sm.push(c1);    sm.push(c21);   sm.push(c31);
34         sm.push(c32);   sm.push(c22);   sm.pop(r22);
35         sm.pop(r31); // error
36         sm.pop(r31);
37         sm.pop(r21);
38         sm.pop(r1);
39
40         }
```

Figure 5.21: Stack simulation using p-typestate

presented in this chapter will be another useful addition.

5.9 Chapter Summary

In this chapter, we presented a definition of regular typestates and the motivation for a more expressive notion of typestates. Following this, we introduced the idea of Parameterized typestates (p-typestates) and formalized the concept. We presented a way to implement these p-typestates in a typestate-oriented programming language using dependent types. We discussed the challenges associated with the p-typestate type system, typechecking, and our approach to handling these challenges. We concluded the chapter by presenting a novel and simple loop invariant calculation technique and its integration to our language. We empirically showed the use cases for our language and invariant calculation.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

Analyzing and verifying behavioral properties of programs is a challenging problem. The challenges further surge with rising complexity of programs, with additions of newer programming features. On the other hand, these behavioral properties can themselves become increasingly complex, leaving current algorithms, tools and systems to verify the dynamic behavior of programs, ineffective. Type systems are the most prevalently used static, light-weight verification systems for verifying certain properties of programs. Unfortunately, simple types are inadequate at verifying many behavioral/dynamic properties of programs. Typestates can tame this inadequacy of simple types, by associating each type in a programming language with a state information. This state is an abstraction which captures the behavior of the program and aids in reasoning about it.

There are two major challenges in statically analyzing and verifying typestate properties over programs. The first one is the increasing complexity of programs. The original work on typestates can only verify/analyze a typestate property over very simple programs, lacking dynamic memory allocation and aliasing. Subsequently, the following works on typestates extended and improvised the analysis over programs with aliasing and dynamic memory. However, the state-of-the-art static typestate analysis works still cannot handle formidably rich programming features like asynchrony, library calls and callbacks, concurrency, etc.

The second challenge is in handling the nature of the properties being verified. The original and the current notion of typestates can only verify a property definable through a finite-state abstraction. This makes the state-of-the-art typestate analysis and verification works inadequate to verify useful but richer non-regular program properties. For example, using classical typestates we can verify a property like, pop be called on a stack only after a push

operation, but we cannot verify a nonregular program property like, number of push operations should be at least equal to the number of pop operations. Another example is that of a well-formed XML file which must have matching opening and closing XML tags. Currently, these behavioral properties are mostly verified/enforced by programmers at runtime via explicit checks. Unfortunately, these runtime checks are costly, error-prone, and lay an extra burden on the programmer.

Towards handling the first challenge, we develop an asynchrony-aware typestate analysis over Android Inter-Component Control Flow Graph (AICCFG), an intermediate program representation for Android applications. The AICCFG soundly models the asynchronous control flow semantics (single threaded, non-preemptive) of ICC, the associated lifecycle of components, and interactions between them. The asynchrony-aware static analysis and formal modeling of Android control flow is a small yet significant step towards a sound modeling and static typestate (and other static analyses) analysis for Android applications. This problem of building a sound and practically precise static analysis for Android applications is still an unsolved problem, even after great interest shown recently from the research community. There have been a few earlier attempts at formal modeling of Android applications, but none have focused towards capturing the correct asynchronous control flow semantics in these applications before this thesis. Our modeling and the asynchronous static analysis idea is not limited to Android applications, as the programming model and the control flow features of Android discussed in the thesis are shared by other event-based programs like iOS applications, web browser extensions applications, etc. Thus our ideas, model and analysis are easily extensible towards other similar programs and programming models.

Towards handling the other challenge, we develop a typestate-oriented programming language incorporating parameterized typestates (p-typestate). This allows us to create correct by construction imperative programs guaranteeing useful p-typestates properties. Statically verifying real-world programs against rich non-regular behavioral properties/contracts, like "number of messages sent over a channel, must be always greater than or equal to the number of messages received", is one of the long-term goals of programming languages and verification community. The verification community has made certain strides towards this goal through works on verifying infinite state systems. Unfortunately, these works are mostly applicable to abstract models of the programs rather than real-world programs. Further, these models have to be provided by the user or the programmer using these verification techniques. This makes these verification techniques inadequate to efficiently verify real-world programs.

The programming language community has been even more moderate in its movement towards this goal. Types and type systems, which have been the cornerstone of the light-weight

static verification technique for programming languages has mostly been applied to verify and reason about the structural properties of programs (properties remaining constant throughout the lifetime of data). Typestates extended the concepts of simple types with states and state transitions. This makes them adequate to capture a few behavioral properties. Unfortunately, available typestate works prior to this thesis have been limited both by their expressiveness and by the lack of tools and techniques for typestate analysis over programs with complex features. This thesis provides solutions to both these limitations. The Parameterized Typestates, the dependent typestate system and the typestate-oriented programming language, provides an expressive and easy way to adopt the language and a subsystem to create correct-by-construction programs, verifying rich non-regular typestate properties. The language subsystem may act as a basis for building systems with automatic statically verified behavioral properties of programs, like the properties defined over sessions between processes, and other session-related and contractual properties.

Automatic checking of rich logical properties and corresponding typechecking for type systems require invariants for loops and recursive structures to be provided by a programmer. This is a big hindrance for programmers to adopt these richer type systems. The p-typestate type system also requires such a loop invariant from the programmer, we placate this burden from the programmers by presenting a novel loop invariant calculation approach for Presburger definable systems.

Overall, this thesis is a significant step towards static verification of behavioral properties of programs. It discusses, the fundamental challenges associated with using the current static analysis and programming languages support for verifying these properties (w.r.t. typestates) and provides both theoretical and practical solutions to these challenges and shows its effectiveness over a rich set of benchmarks.

6.2 Future Work

Verifying behavioral properties of programs in a light-weight, automatic, and modular fashion is a challenging and popular research problem. In this thesis, we have taken a small step towards this goal of generating tools, techniques and programming language support to develop programs with guaranteed behavioral properties. There are various possible directions for future work, we discuss a few in this section.

Consider again the *complexity of programs vs. richness of properties* graph discussed in Chapter 1. Extending our thesis along both the axes leads to a few useful possible future works. Along the y-axis, we can first extend our asynchrony-aware static analysis approach with concurrency. A large percentage of Android applications are multi-threaded, extending

the analysis approach to handle multi-threading control and data flows can be a challenging and interesting problem as none of the state-of-the-art static analysis works for Android applications handle multi-threading in a sound and practically precise way. Similar to this, *Java reflection* is another programming feature which although discussed by various works for Android static analysis yet is still unsolved.

Along the x-axis, looking for logical families more expressive than the Presburger definable logic, such that the type-checking over these still remains decidable can be really useful to verify richer program properties than p-typestate properties. For instance, *Relational Properties*, like defining a relation between two elements in an array or some other structure, cannot be defined using Presburger definable logic. Such a property can be used to statically model functional properties of programs like verifying a sorted array or checking a Binary Search Tree (BST) structure of a BST, etc.

Similarly, extending the thesis above the diagonal will allow developing p-typestate analysis for Android applications. This can be a major extension of the thesis as Android has various program properties like *dynamic permission* granting and revoking which cannot be modeled using finite automata. p-typestates can effectively verify such properties.

The p-typestates system presents a statically typed type system, there can be a possible extension of the p-typestate type system with gradual typing. This will aid in increasing the precision of the verification and will also aid in reducing the burden of annotation on the programmer. A modular p-typestate type system on the lines of Deline et. al. [82] will be really useful.

Another interesting direction of future work will be to extend the p-typestates and core language with richer features like polymorphic types, asynchrony, concurrency, etc. This extension will lead to a further investigation of the connections between Session types [120], normal typestates and p-typestates. This can further aid in extending the p-typestates to concurrent and distributed setting using session types.

A complete formal characterization and applications of the loop invariant calculation approach presented in our paper will be another useful direction for future work.

Appendix

Subtyping Rules for DRIP

Subtyping Subtyping relation in our language is *reflexive* and *transitive*.

$$\text{T-Sub-pair}\frac{\Phi, \Gamma \vdash \tau_1 <: \tau_2 \qquad \tau_3 <: \tau_4}{\Phi, \Gamma \vdash \mathit{pair}(\tau_1, \tau_3) <: \mathit{pair}(\tau_2, \tau_4)}$$

(T-Sub-pair) defines subtyping rule for *pair*(τ_1, τ_2) type is a standard pair wise subtyping

$$\text{T-Sub-formula}\frac{\begin{array}{c}\Phi, \Gamma \vdash \phi_1 : \text{Type},\ \phi_2 : \text{Type} \\ \phi_1 \vDash \phi_2\end{array}}{\phi_1 <: \phi_2}$$

Rule (T-Sub-formula), defines subtyping rule for two Presburger formulas ϕ_1 and ϕ_2. A formula $\phi_1 <: \phi_2$ iff ϕ_1 satisfies ϕ_2, where satisfaction is defined as a standard Presburger formula implication.

$$\text{T-Sub-}p^{x,s}\frac{\begin{array}{cc}\Phi, \Gamma \vdash x_1 : \phi & \Phi, \Gamma \vdash x_2 : \phi \\ \Phi, \Gamma s_1 : S & \Phi, \Gamma s_2 : S \\ \Phi, \Gamma \vdash x_1 <: x_2 & \Phi, \Gamma \vdash s_1 <: s_2\end{array}}{p^{x,s}\{x_1/x, s_1/s\} <: p^{x,s}\{x_2/x, s_2/s\}}$$

Bibliography

[1] Homotopy type theory. https://homotopytypetheory.org/.

[2] Android source. https://source.android.com/. 14, 23

[3] Appbrain. http://www.appbrain.com/stats/number-of-android-apps. 13, 73

[4] Asyncbench, git repo. http://git-for-asyncbench. 120, 121, 122

[5] Buffer overflow. https://www.owasp.org/index.php/Buffer_Overflow.

[6] Android media player. https://developer.android.com/reference/android/media/MediaPlayer.html. 111

[7] Android. https://developer.android.com/studio/index.html. xiii, 13, 23, 26, 115

[8] Ashish mishra-git-p-typestate. https://github.com/ashishtheaegis/p-typestate. 170

[9] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640073. URL http://doi.acm.org/10.1145/1639950.1640073. 5, 9, 52, 60, 61, 130, 139, 140, 153

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75 (2):87–106, November 1987. ISSN 0890-5401. doi: 10.1016/0890-5401(87)90052-6. URL http://dx.doi.org/10.1016/0890-5401(87)90052-6. 71

[11] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. Trex: A tool for reachability analysis of complex systems. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 368–372, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42345-1. URL http://dl.acm.org/citation.cfm?id=647770.734247. 46

[12] Alessandro Armando, Gabriele Costa, and Alessio Merlo. *Formal Modeling and Reasoning about the Android Security Framework*, pages 64–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41157-1. doi: 10.1007/978-3-642-41157-1_5. URL https://doi.org/10.1007/978-3-642-41157-1_5. 51

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594299. URL http://doi.acm.org/10.1145/2594291.2594299. 51, 53, 73

[14] Lennart Augustsson. Cayenne—a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289451. URL http://doi.acm.org/10.1145/289423.289451. 64, 134

[15] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *Proceedings of the Third International Conference on Automated Technology for Verification and Analysis*, ATVA'05, pages 474–488, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29209-8, 978-3-540-29209-8. doi: 10.1007/11562948_35. URL http://dx.doi.org/10.1007/11562948_35. 46, 47, 130, 155, 156, 167, 168

[16] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 378–394, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-69735-0. URL http://dl.acm.org/citation.cfm?id=1763048.1763087. 70

[17] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_10. URL http://dx.doi.org/10.1007/978-3-642-03013-0_10.

[18] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 195–219, 2009. doi: 10.1007/978-3-642-03013-0_10. URL https://doi.org/10.1007/978-3-642-03013-0_10. 5, 60, 62

[19] Andreas Blass and Yuri Gurevich. The underlying logic of hoare logic. In *IN CURRENT TRENDS IN THEORETICAL COMPUTER SCIENCE*, pages 409–436, 1997. 31

[20] Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer*, 14(3):307–326, Jun 2012. ISSN 1433-2787. doi: 10.1007/s10009-010-0183-5. URL https://doi.org/10.1007/s10009-010-0183-5. 5, 62, 63

[21] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR '97, pages 135–150, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63141-0. URL http://dl.acm.org/citation.cfm?id=646732.701281. 46

[22] Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6. doi: 10.1007/978-3-642-03153-3_2. URL http://dx.doi.org/10.1007/978-3-642-03153-3_2. 64, 139

[23] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, and Tomáš Vojnar. Automatic verification of integer array programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 157–172, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_15. URL http://dx.doi.org/10.1007/978-3-642-02658-4_15. 70

[24] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and*

Abstract Interpretation, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-31139-4, 978-3-540-31139-3. doi: 10.1007/11609773_28. URL http://dx.doi.org/10.1007/11609773_28. 70

[25] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500581. URL http://doi.acm.org/10.1145/2500365.2500581. 41

[26] Edwin C. Brady. Idris —: Systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0. doi: 10.1145/1929529.1929536. URL http://doi.acm.org/10.1145/1929529.1929536. 40, 41

[27] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In *Proceedings of the Third International Conference on Concurrency Theory*, CONCUR '92, pages 123–137, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55822-5. URL http://dl.acm.org/citation.cfm?id=646727.703214. 46

[28] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 47–56, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.53995. URL http://doi.acm.org/10.1145/53990.53995. 6

[29] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. ISSN 0360-0300. doi: 10.1145/6041.6042. URL http://doi.acm.org/10.1145/6041.6042. 39

[30] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 147–163, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24297-X, 978-3-540-24297-0. doi: 10.1007/978-3-540-30579-8_11. URL http://dx.doi.org/10.1007/978-3-540-30579-8_11. 70

[31] Sarah Chasins. Efficient implementation of the plaid language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 209–210, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048211. URL http://doi.acm.org/10.1145/2048147.2048211. 5, 60, 61, 130

[32] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*. The Internet Society, 2013. URL http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#ChenJDDMMWRS13. 52

[33] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0643-0. doi: 10.1145/1999995.2000018. URL http://doi.acm.org/10.1145/1999995.2000018. 55

[34] Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs*, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24730-2. doi: 10.1007/978-3-540-24730-2_15. URL https://doi.org/10.1007/978-3-540-24730-2_15. 71

[35] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994. ISSN 0164-0925. doi: 10.1145/186025.186051. URL http://doi.acm.org/10.1145/186025.186051. 46

[36] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00898-5. URL http://dl.acm.org/citation.cfm?id=1765871.1765903. 71

[37] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 268–279, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64608-6. URL http://dl.acm.org/citation.cfm?id=647767.760727. 46

[38] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167 – 177, 1996. ISSN 0167-6423. doi: https://doi.org/10.1016/0167-6423(95)00021-6. URL http://www.sciencedirect.com/science/article/pii/0167642395000216. 34

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973. 31, 70, 155

[40] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM. doi: 10.1145/512760.512770. URL http://doi.acm.org/10.1145/512760.512770. 70

[41] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368127. URL http://doi.acm.org/10.1145/1368088.1368127. 71

[42] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. *Privilege Escalation Attacks on Android*, pages 346–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-18178-8. doi: 10.1007/978-3-642-18178-8_30. URL https://doi.org/10.1007/978-3-642-18178-8_30. 51, 55

[43] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24. 34, 43, 151

[44] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, pages 139–145, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-56517-5. URL http://dl.acm.org/citation.cfm?id=645891.671431. 40

[45] David A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, Inc., New York, NY, USA, 1991. ISBN 0-471-92784-8. 46

[46] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924971. 51, 73

[47] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302467. URL http://doi.acm.org/10.1145/302405.302467. 71

[48] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. doi: 10.1145/337180.337240. URL http://doi.acm.org/10.1145/337180.337240. 71

[49] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 232–247, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67770-4. URL http://dl.acm.org/citation.cfm?id=647769.734087. 46

[50] R Fagin and M Y Vardi. An internal semantics for modal logic. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 305–315, New York, NY, USA, 1985. ACM. ISBN 0-89791-151-2. doi: 10.1145/22145.22179. URL http://doi.acm.org/10.1145/22145.22179. 31

[51] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. *SIGPLAN Not.*, 37(5):13–24, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512532. URL http://doi.acm.org/10.1145/543552.512532. 153

[52] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17 (2):9:1–9:34, May 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL http://doi.acm.org/10.1145/1348250.1348255. 111

[53] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17 (2):9:1–9:34, May 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL http://doi.acm.org/10.1145/1348250.1348255. 4, 5, 6, 44, 58, 60, 130, 139

[54] Alain Finkel and Jérôme Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In *Proceedings of the 22Nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, FST TCS '02, pages 145–156, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00225-1. URL http://dl.acm.org/citation.cfm?id=646840.708796. 46, 47, 155, 156, 158, 159, 167, 176

[55] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41791-5. URL http://dl.acm.org/citation.cfm?id=647540.730008. 71

[56] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558. URL http://doi.acm.org/10.1145/512529.512558. 61, 63, 64, 130, 134

[57] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. *Modeling and Enhancing Android's Permission System*, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33167-1. doi: 10.1007/978-3-642-33167-1_1. URL https://doi.org/10.1007/978-3-642-33167-1_1. 51

[58] Adam Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. 2009. 52, 55

[59] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3):34:1–34:51, January 2014. ISSN

0360-0300. doi: 10.1145/2506375. URL http://doi.acm.org/10.1145/2506375. 70

[60] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. *ICE:ARobustFrameworkforLearningInvariants*, pages 69–87. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9_5. URL https://doi.org/10.1007/978-3-319-08867-9_5. 71

[61] Simon Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida. Theory and Applications of Behavioural Types (Dagstuhl Seminar 17051). *Dagstuhl Reports*, 7(1): 158–189, 2017. ISSN 2192-5283. doi: 10.4230/DagRep.7.1.158. URL http://drops.dagstuhl.de/opus/volltexte/2017/7249. 2, 3, 130

[62] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 430–440, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070542. URL http://dx.doi.org/10.1109/ICSE.2009.5070542. 71

[63] Chris Hawblitzel. Linear types for aliased resources. Technical report, October 2005. URL https://www.microsoft.com/en-us/research/publication/linear-types-for-aliased-resources/.

[64] Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrei Voronkov. Invariant and type inference for matrices. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 163–179, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11318-4, 978-3-642-11318-5. doi: 10.1007/978-3-642-11319-2_14. URL http://dx.doi.org/10.1007/978-3-642-11319-2_14. 70

[65] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 132–146, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7. doi: 10.1145/320384.320395. URL http://doi.acm.org/10.1145/320384.320395. 52, 96, 140

[66] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented*

Programming, Genoa, pages 27–51, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_3. URL http://dx.doi.org/10.1007/978-3-642-03013-0_3. 5

[67] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. Symdroid: Symbolic execution for dalvik bytecode. 2012. 51

[68] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. POPL '07, pages 339–350. ACM, 2007. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190266. URL http://doi.acm.org/10.1145/1190216.1190266. xii, 55, 57, 70, 74, 75, 83, 90, 92, 111, 115, 118, 120

[69] V. Kodaganallur. Incorporating language processing into java applications: a javacc tutorial. *IEEE Software*, 21(4):70–77, July 2004. ISSN 0740-7459. doi: 10.1109/MS.2004.16. 167

[70] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 470–485, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00592-3. doi: 10.1007/978-3-642-00593-0_33. URL http://dx.doi.org/10.1007/978-3-642-00593-0_33. 70

[71] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. *Proving Safety with Trace Automata and Bounded Model Checking*, pages 325–341. Springer International Publishing, Cham, 2015. ISBN 978-3-319-19249-9. doi: 10.1007/978-3-319-19249-9_21. URL https://doi.org/10.1007/978-3-319-19249-9_21. 71

[72] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59(1-2):157–180, July 1988. ISSN 0304-3975. doi: 10.1016/0304-3975(88)90100-4. URL http://dx.doi.org/10.1016/0304-3975(88)90100-4.

[73] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 493–508, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_37. URL http://dx.doi.org/10.1007/978-3-642-02658-4_37. 70

[74] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999. ISBN 978-1-4615-5229-1. doi: 10.1007/978-1-4615-5229-1_12. URL http://dx.doi.org/10.1007/978-1-4615-5229-1_12. 4, 130, 132

[75] Jérôme Leroux and Gérald Point. Tapas: The talence presburger arithmetic suite. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 182–185, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00767-5. doi: 10.1007/978-3-642-00768-2_18. URL http://dx.doi.org/10.1007/978-3-642-00768-2_18. 156

[76] L. Li, A. Bartel, T. F. Bissyand, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, volume 1, pages 280–291, 2015. doi: 10.1109/ICSE.2015.48. 10, 30, 51, 53, 73, 80, 93

[77] Li Li, Tegawend F. Bissyand, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88(Supplement C):67 – 95, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.04.001. URL http://www.sciencedirect.com/science/article/pii/S0950584917302987. 55

[78] Francesco Logozzo. *Automatic Inference of Class Invariants*, pages 211–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24622-0. doi: 10.1007/978-3-540-24622-0_18. URL https://doi.org/10.1007/978-3-540-24622-0_18. 70

[79] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382223. URL http://doi.acm.org/10.1145/2382196.2382223. 73

[80] A. Pnueli M. Sharir. Two approaches to interprocedural data flow analysis. pages 143–152, 1981. 30, 31, 56

[81] Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam Srivas. Accelerating invariant generation. In *Proceedings of the 15th Conference on Formal*

Methods in Computer-Aided Design, FMCAD '15, pages 105–111, Austin, TX, 2015. FMCAD Inc. ISBN 978-0-9835678-5-1. URL http://dl.acm.org/citation.cfm?id=2893529.2893550. 69, 71

[82] Rob DeLine Manuel Fahndrich. Typestates for objects. In *ECOOP 2004 Object-Oriented Programming, 18th European Conference*, volume 3086, pages 465–490. Springer Verlag, June 2004. URL https://www.microsoft.com/en-us/research/publication/typestates-for-objects/.

[83] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45(1):299–312, January 2010. 4, 5, 6, 9, 58, 59, 62, 130, 133, 177, 183

[84] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. 130

[85] Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[86] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. Iterative user-driven fault localization. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing*, pages 82–98, Cham, 2016. Springer International Publishing.

[87] Ernie Cohen, Mark A. Hillebr, Stephan Tobies, Micha Moskal, and Wolfram Schulte. Verifying c programs: A vcc tutorial, 2015. 131, 133

[88] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998. 131

[89] I. McGinniss. *Theoretical and Practical Aspects of Typestate*. University of Glasgow, 2013. URL https://books.google.co.in/books?id=jfkArgEACAAJ. 136

[90] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 5th edition, 2009. ISBN 1584888768, 9781584888765. 143

[91] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814. 139

[92] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967. ISBN 0-13-165563-9. 62, 63

[93] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-306-46394-6. 42

[94] J. Strother Moore. Meta-level features in an industrial-strength theorem prover. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 425–426, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103707. URL http://doi.acm.org/10.1145/2103656.2103707. 39, 68

[95] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 347–366, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449792. URL http://doi.acm.org/10.1145/1449764.1449792.

[96] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 347–366, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449792. URL http://doi.acm.org/10.1145/1449764.1449792. 41

[97] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100. 51

[98] Ulf Norell. *Towards a practical programming language based on dependent type theory*. No: 2677 Technical report Department of Computer Science and Engineering, Chalmers University of Technology and Gteborg University, no: 33. 2007. ISBN 978-91-7291-996-9. 166. 5

[99] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. *SIGPLAN Not.*, 43(10):457–474, October 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449800. URL http://doi.acm.org/10.1145/1449955.1449800. 63

[100] Jeff Paris. A mathematical incompleteness in peano arithmetic**editor's note: Since 1931, the year godel's incompleteness 'theorems were puhlishcd. mathematicians have been

looking for a strictly mathematical example of an incompleteness in first-order peano arithmetic, one which is mathematically simple and interesting and does not require the numerical coding of notions from logic. the first such examples were found early in 1977. when this handbook was almost finished. we are pleased to be able to include a final chapter which nrcscnts the most striking example to date. it is a fitting conclusion since it brings together ideas from all parts of the handbook. In Jon Barwise, editor, *HANDBOOK OF MATHEMATICAL LOGIC*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 1133 – 1142. Elsevier, 1977. doi: https://doi.org/10.1016/S0049-237X(08)71130-3. URL http://www.sciencedirect.com/science/article/pii/S0049237X08711303. 31, 110

[101] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of java generics. *Empirical Softw. Engg.*, 18(6):1047–1089, December 2013. ISSN 1382-3256. doi: 10.1007/s10664-012-9236-6. URL http://dx.doi.org/10.1007/s10664-012-9236-6. 40, 64, 134

[102] Corina S. Păsăreanu and Willem Visser. *Verification of Java Programs Using Symbolic Execution and Invariant Generation*, pages 164–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24732-6. doi: 10.1007/978-3-540-24732-6_13. URL https://doi.org/10.1007/978-3-540-24732-6_13. 40, 69, 131

[103] Etienne Payet and Fausto Spoto. An operational semantics for android activities. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 121–132, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543748. URL http://doi.acm.org/10.1145/2543728.2543748. 42

[104] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. *SIGSOFT Softw. Eng. Notes*, 29(6):23–32, October 2004. ISSN 0163-5948. doi: 10.1145/1041685.1029901. URL http://doi.acm.org/10.1145/1041685.1029901. 52

[105] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098. 70

[106] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288. 49, 96

[107] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-338-9. doi: 10.1145/1572272.1572284. URL http://doi.acm.org/10.1145/1572272.1572284. 71

[108] Mojżesz Presburger and Dale Jabcquette. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12(2):225–233, 1991. doi: 10.1080/01445340910883718 7. URL http://dx.doi.org/10.1080/01445340910883718 7. 31, 32, 38, 39, 46

[109] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, March 2000. ISSN 0164-0925. doi: 10.1145/349214.349241. URL http://doi.acm.org/10.1145/349214.349241. 41

[110] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95, pages 49–61. ACM, 1995. ISBN 0-89791-692-1. doi: 10.1145/199448.199462. URL http://doi.acm.org/10.1145/199448.199462. 71

[111] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, April 2007. ISSN 0747-7171. doi: 10.1016/j.jsc.2007.01.002. URL http://dx.doi.org/10.1016/j.jsc.2007.01.002. 42, 43

[112] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602. URL http://doi.acm.org/10.1145/1375581.1375602. 109

[113] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. *SIGPLAN Not.*, 39(1):318–329, January 2004. ISSN 0362-1340. doi: 10.1145/982962.964028. URL http://doi.acm.org/10.1145/982962.964028. 6, 30, 31, 53, 55, 56, 75, 115

[114] Walter J. Savitch. *Pascal*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 3rd edition, 1987. ISBN 0805383883. 70

[115] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312929. URL http://dx.doi.org/10.1109/TSE.1986.6312929. 10, 64, 68, 131, 153, 166

[116] Rob Strom and Shaula Yemini. The nil distributed systems programming language: A status report. *SIGPLAN Not.*, 20(5):36–44, May 1985. ISSN 0362-1340. doi: 10.1145/988327.988333. URL http://doi.acm.org/10.1145/988327.988333. 70

[117] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.1*, October 2001. http://coq.inria.fr. 39

[118] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 1–11, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224168. URL http://doi.acm.org/10.1145/224164.224168. 3, 4, 5, 31, 43, 57, 74, 75, 110, 130, 137, 139

[119] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. CASCON '99, pages 13–. IBM Press, 1999. 58

[120] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, August 2012. ISSN 0890-5401. doi: 10.1016/j.ic.2012.05.002. URL http://dx.doi.org/10.1016/j.ic.2012.05.002. 64, 130, 134

[121] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[122] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015. ISSN 0001-0782. doi: 10.1145/2699407. URL http://doi.acm.org/10.1145/2699407. 75, 87, 120

[123] David Wakeling. *Linearity and Laziness*. PhD thesis, York, UK, UK, 1991. UMI Order No. GAXDX-93038. 6, 177, 183

[124] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 127–139, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944718. URL http://doi.acm.org/10.1145/944705.944718.

[125] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. CCS '14, pages 1329–1341. ACM, 2014. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660357. URL http://doi.acm.org/10.1145/2660267.2660357. 42

[126] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, LICS '00, pages 375–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0725-5. URL http://dl.acm.org/citation.cfm?id=788022.789021.

[127] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277732. URL http://doi.acm.org/10.1145/277650.277732. 63

[128] Mengwei Xu, Yun Ma, Xuanzhe Liu, Felix Xiaozhu Lin, and Yunxin Liu. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 143–152, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-4913-0. doi: 10.1145/3038912.3052645. URL https://doi.org/10.1145/3038912.3052645. 30, 55, 73, 80, 93

[129] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 400–411, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784766. URL http://doi.acm.org/10.1145/2784731.2784766. 10, 40, 68, 131

[130] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM. 40, 64, 68

[131] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. 55

[132] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 342–363, Berlin, Heidelberg, 2006. Springer-Verlag. 71, 166
130
132