Analysis and Verification of Rich Typestate Properties for Complex Programs

Ashish Mishra[1] and Y. N. Srikant[2]

1 **Department of Computer Science and Automation**
Indian Institute of Science, India
`ashishmishra@csa.iisc.ernet.in`

2 **Department of Computer Science and Automation**
Indian Institute of Science, India
`srikant@csa.iisc.ernet.in`

Abstract

Typestates are useful programming language concepts to model software protocols. In this thesis we provide programming language based approaches for analysis, checking and verification of rich typestate properties over complex programs. Firstly, we use known typestate analysis approaches to capture crucial API protocol violations in Android applications. The complex control flow semantics of these programs makes the task challenging, while the excessive usage of resources and other APIs by Android makes it important. Secondly, we tackle the expressive limitations associated with typestates, and present a generalized notion of typestate using the expressive power of dependent types. These expressive typestates, which we term as Beyond-Regular Typestate (BR-typestate), are expressive enough to model many important non-regular properties (typestates can only express regular program properties), and yet have decidable type-checking, and even a decidable type-inference in certain cases. We further present a practical typestate oriented, dependently typed language incorporating these BR-typestates and present soundness results about the type system. For both the parts of the work, we create prototype systems to empirically evaluate the concepts discussed.

1998 ACM Subject Classification "D.2.4 Software/Program Verification", "F.3.1 Specifying and Verifying and Reasoning about Programs", "F.3.2 Semantics of Programming Languages"

Keywords and phrases Typestate, Dependent types, Type systems, Android Analysis

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Protocols are one of the most commonly used abstract concepts, acting as an implicit contract between components of a software system or across systems. These contracts must be respected by all the software components participating in the protocol. Some of the common examples include a producer-consumer related protocol between two components acting as server and client respectively, or a session-based protocol between the sender and the receiver. Often these protocols need not be a multi-party contract, rather a set of rules associated with a data structure or an object. For example, the initialization operation is only allowed on an uninitialized variable, a File object can be read only if it has been opened, *etc.* Violating these protocols could lead to semantically invalid programs, or in some cases might open exploitable vulnerabilities in programs. For instance, *failing to check array bounds* might lead to buffer-overflow vulnerabilities.

Typestates [18] were introduced by Strom and Yemini as a programming language concept to capture the state associated with a type. Typestates allows programmers to model and

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–9
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

statically check stateful protocols and contracts. For example, a File type can have two different typestates *viz. open* and *close*, with a protocol allowing read operation only in the *open* state.

1.1 Goal of the Thesis

In this thesis, we present programming language based approaches for analysis, checking and verification of rich typestate related properties over complex programs. In the first part, we present a sound typestate analysis over asynchronous, event based programs like Android applications. Through this we check API protocol violations and other security protocol violations in Android applications. The analysis necessitates to correctly model the control flow semantics of Android applications with its intricacies, such as asynchronous calls, multiple entry-points and control flow semantics enforced by the Android framework.

Typestates can enforce conceptual properties over programs and aid in making software reliable and robust. However, they can only express program properties from the regular language domain. For instance, typestates can enforce regular program properties like *Pop be called on a stack object only after a call to Push* or *Pop is never called on an empty stack*, but cannot enforce a non-regular property like *Push be called at least as many times as Pop*.

In the second part of the thesis we work at solving this expressive limitation of typestate. We propose a generalized notion of typestate which we call "Beyond-Regular Typestate (BR-typestate)". BR-typestates are restricted, dependently typed extension of the normal typestate. They enjoy richer expressiveness of dependent types, yet preserve the decidability of type-checking, unlike general dependent types where type-checking is undecidable. Further, to make them practically useful we present a dependently typed, typestate-oriented programming language, whose type system incorporates these BR-typestates making it expressive enough to implement and check programs requiring non-regular program properties which could be statically type-checked in an efficient way.

2 Asynchrony-Aware static analysis for Android Applications

Android applications have convoluted control flow with asynchronous inter-component communication, library callbacks, event handling and other Android framework enforced control flow semantics. A sound static analysis of such programs requires a correct modeling of Android control flow, failing to model this leads to both unsound results and loss in precision in many cases. For instance, consider a `FileReader` application in figure 1. The application has two activities `SelectActivity` and `ReadFileActivity`. The application allows the user to select a file in `SelectActivity` and open the file in `ReadFileActivity`. The `FileReader` object, line 2, is a global static reference which is accessible through both the components. To verify a typestate property like- *The application never reads from a closed FileReader*, the analysis needs to verify and guarantee that the `FileReader.read()` is never called when the `FileReader` has been closed using `FileReader.close()` and not reopened again. To soundly capture this, the analysis must treat the inter-component call at line 11 as asynchronous call. Further, the analysis should have a correct modeling of the life-cycle of an Android `Activity` component, which enforces that `onResume()` of `SelectActivity` is executed before the dispatch of the pending asynchronous call to `ReadFileActivity`. Incorrect modeling of these control flows will lead to missing of typestate violation possibly occurring at line 36.

Our main idea in this part of the work is to solve this unsoundness problem by presenting an Android inter-component control flow graph (AICCFG). This is an intermediate program

```
class SelectActivity extends ActionBarActivity{
 public static FileReader myFileReader;
 protected void onCreate(Bundle savedInstanceState){
 (...)
 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 block
 }
 (...)
 protected void onResume(){
   (...)
   try{
     myFileReader.close();
   }//catch block
 }
}
```

```
class ReadFileActivity extends ActionBarActivity {
 (...)
 protected void onStop(){
   (...)
    ...
   int data = SelectActivity.myFileReader.read();
   Log.d("ReadFileActivity", "data " +data);
  }
}
```

Figure 1 FileReader Application

representation for Android applications. It soundly captures the control-flow semantics involving asynchrony and other complex features, like framework callbacks and life-cycle of components. We further build an asynchronous inter-procedural typestate analysis over the AICCFG created for the application.

2.1 Asynchronous Inter-component Control Flow Graph (AICCFG)

An AICCFG is an asynchronous control flow graph $G_* = (V_*, E_*)$ for an Android application, modeling the asynchronous calls, event handlers and lifecycle callbacks invoked by the Android framework. An AICCFG acts as a sound intermediate program representation for applications. Figure 2 shows a part of the AICCFG for the example FileReader application in figure 1. The AICCFG has a special dispatch node (gray node vd), which manages the asynchrony and other life-cycle related control flow features. Interested readers can find the details in [13]. We first present an algorithm to generate such an AICCFG for a given application. We then build a flow sensitive version of an asynchronous typestate analysis, which is an extension of the AIFDS [8] approach, extended for Android applications. The AICCFG generated acts as the input program to this analysis. Further, since ours is the first asynchrony-aware static analysis work over Android applications, we compare it against the state-of-the-art synchronous-only static analyses tools [10, 20].

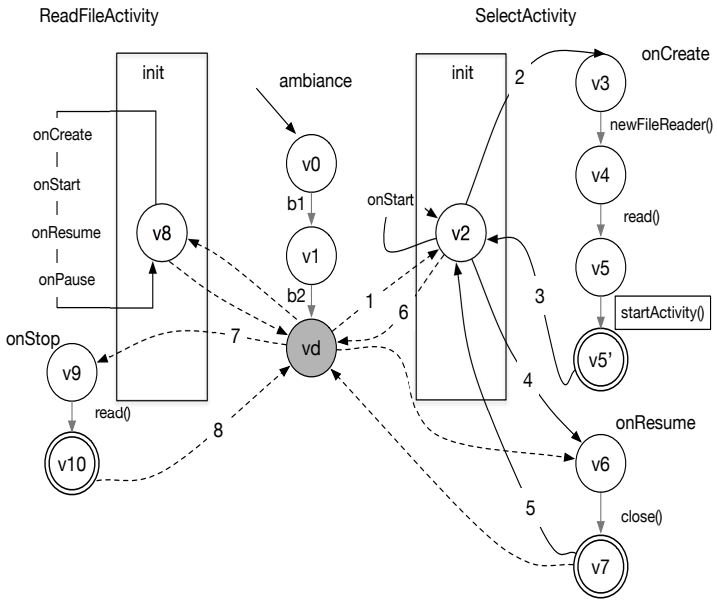

Figure 2 Part of AICCFG for FileReader application

2.2 Evaluation

The major goals of our experimental evaluation are: firstly, to show the the soundness of asynchronous control flow modeling of Android applications. Secondly, to illustrate the usefulness of the modeling, and finally, to compare it against the current works which model these semantics incorrectly. To achieve first and second goals, we present a typestate analysis build on our AICCFG. A sound typestate analysis requires us to maintain a global state of shared resources. Since these global states may change between the time when an asynchronous call is made, and it is actually dispatched by Android framework, we need to correctly capture the asynchronous semantics of these calls. Using our model, and an asynchrony-aware analysis built over it, we find typestate violations in number of applications which could not be captured otherwise. Since there are no other works performing typestate analysis over Android applications, a direct comparison is not possible to achieve the final goal. Thus, we compare the control flow model generated by us against those generated by others. For an efficient comparison we extract the intermediate program representation generated by these approaches (analogous to our AICCFG), which is agnostic about the asynchronous nature of calls in Android applications, and build a synchronous version of the typestate analysis over their program representation. Although, the comparison is presented using typestate analysis, a direct comparison of control flow graphs will clearly show the state-of-the-art approaches missing many valid control flow paths. The typestate analysis further extends this comparison by showing that these missing paths will lead to false negatives as well as false positives. We further release a set of benchmark applications, AsyncBench, 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.

2.3 Significance and Related Work

Static Analysis of Android applications and other such systems is an interesting problem. Solving it efficiently is important to create reliable and safe applications and to help developers debug these applications in a better and faster way. There is a significant interest in the static analysis research community [10, 20, 1, 11] towards analysing these applications for various data and control flow analysis problems ranging from information flow, slicing, permission usability, etc. Most of these analyses emphasize at applying the known static analysis techniques for Java programs to Android applications while focusing on scalability, coverage and efficiency. Unfortunately none of the current static analyses works for Android applications aim at soundly modeling the asynchronous control flow semantics of these applications, and treat asynchronous calls similar to synchronous ones. This causes them to miss many valid control flow paths in the application, thus effecting the correctness of the static analysis. Furthermore, the asynchronous call semantics coupled with the life-cycle semantics enforced by the Android framework, makes the control flow difficult to reason statically and failing to model it further leads to unsound as well as imprecise results. There are several other works on general static typestate analysis, such as [18, 5]. These works tackle challenging problems like capturing typestate changes in presence of aliasing, but they are different than our typestate analysis work which focuses on developing a typestate analysis for Android applications. Thus, while we too handle aliasing in a limited way through intra-procedural alias analysis prior to the main typestate analysis, the challenges in our analysis are more specific to Android applications, like asynchrony, life-cycle modeling, etc.

3 Beyond-Regular Typestate for Non-regular Properties of the Programs

Typestates are useful programming language concept to capture varied program properties, some of which we discussed in first part of the problem. For instance, using typestate, we can specify and check a program property on a Stack data type, like "The pop operation is called only after a push operation is already called". This helps programmers to identify and eliminate a class of semantic bugs in the program. A static type system capable of providing typestate guarantees further aides the elimination of these bugs early in program development.

Although interesting and efficient at capturing the properties related to the state of data, typestates can only express program properties from the regular language domain.

For instance, consider a commonly occurring non-regular property *the property of matching parenthesis*. Figure 3 presents the XMLParser code from the XMLParser library for Java. The parser needs to check the well formedness of xml files, which requires matching opening and closing elements. This is a classic example of a context free language (Dyck language with m parenthesis or D_m). The parser dynamically maintains a stack of opening and closing of elements and checks the well formedness at run-time in lines 7, 8, 9. Since the property of *matching parenthesis* belongs to context free languages, current typestate definition lacks expressiveness to model and check it statically. Moreover, these runtime checks are both costly and error prone and might be difficult to debug as the manifestation of the error may be distant from the cause, both in terms of location and time.

Many important semantic properties of programs do not belong to regular language domain and hence are beyond the expressive capabilities of typestate. A few interesting examples include properties related to producer-consumer, for instance *the number of items*

```
protected int scanEndElement() throws IOException, XNIException {
      (...)
       // pop context
       QName endElementName = fElementStack.popElement();

       String rawname = endElementName.rawname;
       if (!fEntityScanner.skipString(endElementName.rawname)) {
            reportFatalError("ETagRequired", new Object[]{rawname});
       }

       // end
       (...)
       if (fDocumentHandler != null ) {
           fDocumentHandler.endElement(endElementName, null);
       }
       if(dtdGrammarUtil != null)
           dtdGrammarUtil.endElement(endElementName);

       return fMarkupDepth;
 }
```

Figure 3 Example XML Parser checking well formedness of xml files

produced by the producer on the Channel is less than or equal to the number of items consumed by the Consumer process, or properties defined over the size of Arrays and List, for instance, *statically guaranteeing safe array bounds, removing the need for runtime checks*, etc. Thus, there is a need to look further to bridge this gap between important program properties and the expressiveness of typestates, and develop a generalized more expressive notion of Typestate. In the lack of such a typestate, these properties are either checked at runtime [9], or expressed using an expressive but undecidable systems [6, 19, 15], or are converted to abstract model and then verified using existing non-regular verification techniques. The main idea of this part of our work is to attack this gap, we use restricted dependent types with dependent terms belonging to Presburger formulas to capture these non-regular program properties in our typestate. Further, to make them practically usable we propose a dependently typed, typestate oriented programming language, whose type system incorporates these typestates, making it expressive enough to implement and statically check programs requiring non-regular program properties, and yet having a decidable and efficient type-checking.

3.1 BR-Typestate and the Core-language

BR-Typestate is a generalized notion of typestate, it is a dependently typed extension of regular typestate. The dependent terms of the type system belong to a decidable logical fragment of Presburger formulas. We present a core-language with BR-typestate support, which is built over Plaid [2], a typestate oriented programming language. Figure 4 shows a portion of the safe version of Stack data structure written in our language. This implementation guarantees a property, *number of items pushed into a stack are always greater than or equal to the number of items popped*. Along with many other features, it allows users to define dependent type families (line 2), which lets programmer capture runtime properties of data using static types. Further, it allows to instantiate a type family (line 3), which lets the programmer introduce a concrete term of the dependent type family. It also allows programmer to annotate each method with a Hoare style *pre* and *post* conditions using Presburger formulas (lines 5 and 9). This lets programmer capture allowed typestate transitions. The type-checker generates constraints using these annotations and finally passes them to a Presburger constraint solver. The language has a decidable type-checking

```
state StackCheck{
    type ValidStack : Pi (npush, npop | npush >= npop) -> Stack;
    var unique ValidStack (0, 0) -> Stack testStack = new Stack (0, 0);

    method void safePush(var elem) [ValidStack(m, n | m >= n -> Stack >> ValidStack(m', n
         ' | m' = m + 1, n' = n, m' >= n') -> Stack this)]{
        // body
    }

    method int safePop() [ValidStack(m, n | m > n -> Stack >> ValidStack(m', n' | m' = m,
         n' = n+1, m' >= n') -> Stack this)]{
      // body
    }

}
```

Figure 4 Code snippet for safe Stack using BR-typestate

due to the decidability of Presburger arithmetic. Type annotations could be omitted for simple cases. For instance, programmer can omit type annotations for data involved in some primitive arithmetic operations. BR-Typestate system assumes that the **while** syntax is annotated with a loop invariant, and we assume that this is provided by the programmer. This assumption is essential to guarantee termination of our typechecking algorithm. This could be a hard task for a novice programmer, and challenging even for an experienced programmer. Fortunately, this burden could be placated in certain special subclasses of programs or properties for which loop invariants could be effectively inferred. The loop invariant inference is based on the efficient and decidable verification results [4, 3, 7] for some known subclasses of multiple counter machines, like *Flat Counter Machine* [4]. Inferring these loop invariants automatically for general class of programs is a challenging problem, and is left for future work. Details of the language and other results could be found in [14].

3.2 Evaluation

To evaluate the effectiveness of BR-Typestate, we implement some important real world programs requiring static checking of non-regular program properties. These properties could not be enforced using regular typestates. For instance, we implement a statically verified XML parser, and a statically verified static analyser which calculates a set of CFL-reachable inter-procedurally valid paths in the control flow graph of an input program. We also provide safe-versions of Plaid libraries with BR-annotated data types which provide safety guarantees against various non-regular program properties.

3.3 Related Work

In this section we discuss the most closely related works to BR-typestate, and how they are insufficient to address the problem described above. There are works presenting languages incorporating typestate as first class language construct [2], or a type system to capture typestate properties [12]. Although, they allow programmers to develop systems with correct typestate properties, they face the wall of expressive limitations of normal typestate. In absence of the needed expressiveness, they need to revert to runtime checks to check these properties.

Fully dependently typed languages like Coq and Agda [19, 15] provide expressiveness to model rich program properties, but this expressiveness comes at the cost of an undecidable type-checking. One possible approach to tackle this undecidability is to restrict the dependent terms of these languages to belong to a decidable theory. This is semantically equivalent

to defining a language similar to ours in Coq or Agda. This definition will be verbose and complex for a programmer to build. Further, it will still be unsound and the onus will be on programmer to prove the decidability and soundness of this restricted subset in Coq and Agda. Finally, there are works which are some dependently typed extensions for simpler languages [22, 21, 17, 16], similar to our approach. Our work differs from these in terms of the domain of the language, and decidability of type-checking and type inference. For instance, the constrained type of X10 [16] is related to the dependent type language we provide in our work. X10 allows to define "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 plugin. 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 explicit restriction imposed over the dependent terms in our language (Presburger formula language), these restrictions yields a decidable type-checking for all valid programs in our languages and also yields a decidable type-inference in certain special cases. Compared to this, the constraint checking and hence the type-checking is undecidable in general for X10. Further, we provide a formal study of our BR-typestate and discuss its correctness and other formal guarantees, X10's does not look into these aspects of its type system.

4 Conclusion

In conclusion, in this thesis we made a small contribution towards analysing and verifying rich typestate properties over complex programs. We discussed two important typestate related works. Further, BR-typestates could also be coupled with asynchrony-aware modeling and analysis, to verify non-regular program properties over Android applications requiring asynchrony semantics. For example, we can typecheck matching granting and revocation of dynamic *URI Permissions* in applications. We leave such an analysis as a possible future work.

References

1. 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 PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.
2. Sarah Chasins. Efficient implementation of the plaid language. In OOPSLA '11, pages 209–210, New York, NY, USA, 2011. ACM.
3. Hubert Comon and Véronique Cortier. Flatness is not a weakness. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 262–276, London, UK, UK, 2000. Springer-Verlag.
4. Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In CAV '98, pages 268–279, London, UK, UK, 1998. Springer-Verlag.
5. 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.

6 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

7 Oscar H. Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan, and Richard Kemmerer. *Counter Machines: Decidable Properties and Applications to Verification Problems*, pages 426–435. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

8 Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. POPL '07, pages 339–350. ACM, 2007.

9 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

10 L. Li, A. Bartel, T. F. Bissyandé, 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.

11 Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

12 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.

13 A. Mishra, A. Kanade, and Y. N. Srikant. Asynchrony-aware static analysis of android applications. In MEMOCODE '16, pages 163–172, Nov 2016.

14 Ashish Mishra and Y. N. Srikant. Beyond-regular typestate. *CoRR*, abs/1702.08154, 2017.

15 Ulf Norell. *Towards a practical programming language based on dependent type theory*. Doktorsavhandlingar vid Chalmers tekniska högskola. Ny serie, no: 2677Technical report D - Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, no: 33. Institutionen för data- och informationsteknik, Datavetenskap, Programmeringslogik (Chalmers), Chalmers tekniska högskola,, 2007. 166.

16 Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In OOPSLA '08, pages 457–474, New York, NY, USA, 2008. ACM.

17 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

18 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.

19 The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.1*, October 2001. http://coq.inria.fr.

20 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.

21 Hongwei Xi. Imperative programming with dependent types. In LICS '00, pages 375–, Washington, DC, USA, 2000. IEEE Computer Society.

22 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.