

Characterization and Verification of Correctness Criteria for Event-Driven Asynchronous Programs

by

Burcu Özkan

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of
Doctor of Philosophy

in

Computer Engineering



September, 2016

**Characterization and Verification of Correctness Criteria for
Event-Driven Asynchronous Programs**

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Burcu Özkan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Assoc. Prof. Serdar Taşiran

Assist. Prof. Didem Unat

Assoc. Prof. Alper Şen

Prof. Attila Gürsoy

Assist. Prof. Cemal Yılmaz

Date: _____



To my family

ABSTRACT

Asynchronous programming is ubiquitous in modern event driven web and mobile applications for which responsiveness to user inputs is a key concern. These programs are prone to concurrency errors due to the different control flow of asynchronous execution and the inherent scheduling nondeterminism of multithreading. Though recent work focuses on detecting bugs in event driven and asynchronous programs, there is no high-level correctness criteria that is independent of the computing platform or the application logic.

The goal of this dissertation is the characterization of the correctness of event-driven asynchronous programs and building techniques and tools for detecting the concurrency bugs in these programs.

Towards formulating a correctness criteria, we build static and dynamic concurrency exploration tools for asynchronous programs. We devise a sequentialization-based technique which encodes a particular subset of possible concurrent program behaviors into a nondeterministic sequential program, where this subset of behaviors is specified by some bounding parameters. By taking the program synchronization into account, our method provides an efficient exploration that can capture a larger set of concurrent program behaviors for a given bounding parameter.

We also develop a dynamic exploration tool to detect and reproduce asynchrony bugs in Android applications. It adopts systematic concurrency exploration that is well-known for multithreaded programs to event-driven asynchronous programs. Given a sequence of user inputs to an application, the tool systematically executes alternate schedules of tasks in the execution of this sequence of event handlers.

Based on our experience on the analysis of concurrency bugs in asynchronous programs, we define a correctness criteria, *robustness against asynchrony*. It captures

the programmer expectations about the behavior of an event-driven asynchronous program and enables the programmer to ignore certain interference among some parts of his program. Roughly, regardless of the asynchrony and the concurrent accesses to the memory, robustness requires a program to exhibit the same set of behaviors with an abstract view of its execution in which the event handlers and asynchronous invocations run in isolation.

Robustness against asynchrony can be formulated as a conjunction of two intuitive properties: *event serializability* and *event determinism*. This formulation of robustness is important as it yields an efficient algorithmic checking method. Event serializability is an adaptation of the serializability property used in some other contexts to the event handlers in a program. It requires the executions of a sequence of event handlers to be equivalent to a serial execution of them in which an event handler is dispatched only after the completion of all asynchronous tasks of the previous event. Event determinism focuses on a single event handler and requires it to produce a deterministic outcome regardless of the interleaving execution of its asynchronous invocations. We present novel algorithms for the verification of event determinism and event serializability. Remarkably, our method reduces the analysis of the event-driven asynchronous programs with unbounded number of concurrent tasks to the reachability problems in sequential programs.

Our experimental work on the open source Android applications shows that our robustness checking algorithm is viable and can be used in practice. Moreover, the real concurrency bugs in the applications can be captured as robustness violations and unknown bugs can be uncovered by checking the robustness of an application.

ÖZETÇE

Asenkron programlama, kullanıcı girdilerine hızlı yanıt vermenin önemli bir gereksinim olduğu olaya dayalı modern web ve mobil uygulamalarda yaygın olarak kullanılmaktadır. Bu programlar, asenkron program çalışmasının getirdiği farklı kontrol akışı ve çok iş parçacıklı programların doğasında olan zorluklardan ötürü eşzamanlılık hatalarına yatkındırlar. Yakın zamanda, olaya dayalı ve asenkron programlardaki program hatalarını yakalamayı amaçlayan çalışmalar yapılmış olsa da, bu programlar için çalışma platformundan ve uygulamadan bağımsız, üst seviyeli bir doğruluk kriteri henüz tanımlanmamıştır.

Bu tezin amacı, olaya-dayalı asenkron programlar için bir doğruluk kriterinin tanımlanması ve bu programlardaki eşzamanlılık hatalarının belirlenmesinde kullanılacak yöntem ve araçlar geliştirilmesidir.

Bu doğrultuda, asenkron programların eş zamanlılık davranışlarını inceleyen statik ve dinamik araçlar geliştirdik. Eşzamanlı bir asenkron programın çalışma davranışlarının bazı kısıtlama parametrelerine göre belirlenmiş bir alt kümesini, non-deterministik, tek iş parçacıklı bir programda kodlamayı sağlayan yöntemler üzerinde çalıştık. Geliştirdiğimiz yöntem, programda var olan senkronizasyonu göz önüne alarak verilen bir kısıtlama parametresi için daha çok eşzamanlı program davranışını inceleyebilmeye olanak sağlamaktadır.

Bir başka çalışma olarak, Android uygulamalarındaki asenkroni hatalarını yakalayan dinamik bir araç geliştirdik. Bu araç, çok iş parçacıklı programların incelenmesinde bilinen bir teknik olan sistematik eşzamanlılık taramasını olaya-dayalı asenkron programlara uyarlamaktadır. Araç, verilen bir dizi kullanıcı girdisi için, uygulamadaki olay işleme prosedürlerinin oluşturduğu asenkron görevlerin farklı çalışma sıralarını sistematik olarak test etmektedir.

Asenkron programlardaki eşzamanlılık hatalarını incelemede elde ettiğimiz deneyime dayanarak, “asenkrone karşı dayanıklılık” olarak isimlendirdiğimiz bir doğruluk kriteri tanımladık. Bu kriter, programcının olaya-dayalı asenkron programın çalışma davranışları konusundaki beklentilerini karşılamakta ve programın bazı kısımları arasındaki etkileşimi göz ardı edebilmesine izin vermektedir. Kabaca asenkron karşı dayanıklılık, program içindeki asenkron ve programın farklı parçalarının paylaşılan bellek üzerinde girişikleme yaparak işlem yapmasına bakılmaksızın programın tüm olay işleme prosedürlerinin ve tüm asenkron görevlerinin izole bir şekilde çalıştığı soyut bir görünümü ile aynı davranışları sergilemesini gerektirmektedir.

Asenkron karşı dayanıklılık, iki sezgisel özelliğin; *olay serileştirilebilirliği* ve *olay determinizminin* birleşimi olarak ifade edilebilir. Kriterin bu şekilde ifade edilebilmesi, algoritmik olarak verimli bir şekilde kontrol edilebilmesine olanak sağladığı için önemlidir. Olay serileştirilebilirliği, farklı ortamlarda da kullanılan *serileştirilebilirlik* kavramının olay işleme prosedürlerine uyarlamasıdır. Bu özellik, bir dizi olay işleme prosedürünün eşzamanlı çalışmasının, olay işleme prosedürlerinin seri olarak, bir olayın işlenmesi tamamen bittikten sonra bir başka olay işlenerek art arda çalışmasına denk olmasını gerektirmektedir. Olay determinizmi tek bir olay işleme prosedürüne odaklanır. Bir olay işleme prosedürünün, oluşturduğu asenkron görevlerin paylaşılmış bellek üzerinde girişikleme yaparak çalışmasına bakılmaksızın tek deterministik sonucu olmasını gerektirmektedir. Olay serileştirilebilirliği ve olay determinizmi özelliklerinin doğrulanması için yeni yöntemler geliştirdik. Sunduğumuz yöntemin dikkat çekici yanı, eşzamanlı çalışabilen sınırsız sayıda asenkron görev içeren çok iş parçacıklı programların analizini tek iş parçacıklı programlarda ulaşılabilirlik problemine indirgemesidir.

Açık kaynaklı Android uygulamaları üzerinde yaptığımız deneysel çalışmalar, asenkron karşı dayanıklılık analizinin pratikte uygulanabilir olduğunu göstermektedir. Ayrıca, uygulamalardaki gerçek eşzamanlılık hatalarının asenkron karşı dayanıklılık ihlali olarak yakalanabilmesi ve önceden bilinmeyen bazı eşzamanlılık hatalarının da bulunması önerdiğimiz doğruluk kriterinin geçerliliğini göstermektedir.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my supervisor *Serdar Taşırın* for his guidance, patience and support throughout my study. I am lucky to have an inspirational supervisor who has been both an academic advisor and also a mentor. I will always recall his motivational advice which encouraged me during the tough times of my study. I appreciate all his contributions of time and ideas to both my PhD work and my long-term career goals. I would also like to thank his wife *Valerie Taşırın* for her patient comments and advice on academic English writing.

I am fortunate to collaborate with *Ahmed Bouajjani*, *Constantin Enea* and *Michael Emmi* during my PhD work. I am very grateful to them for their mentorship which is not limited to my research visits but throughout my work. I learned a lot from our discussions in my research visits to University Paris-Diderot and my research internship in IMDEA Software Institute. I am very thankful for their valuable contributions, patience to my questions and advice.

I would like to thank *Marc Shapiro* who guided me during my research visit to University Pierre and Mary Curie (UPMC), INRIA and the other collaborators in the FP7 project Syncfree, for their comments and discussions.

I would like to thank to the members of my thesis committee *Didem Unat*, *Alper Şen*, *Attila Gürsoy* and *Cemal Yılmaz* for their valuable comments and feedback on my work. I must also thank to the anonymous reviewers of my publications for their comments and suggestions.

I am grateful to *Selma Tekir* and *Serap Şahin* for their continuous guidance and support. It is impossible to fully express my appreciation for our enjoyable technical and nontechnical discussions.

Former and current members of Research Center for Multicore Software Engineering

(MSRC) *Can, Erdal, Erkut, Hassan, İsmail, Maryam, Süha* and my other office mates *Anıl, Aria, Bakış, Ersan, Houman, İdil, Rustam, Sadra* and *Salih* deserve my special thanks. Thanks to them, we always had a nice atmosphere in our lab, both to discuss our research problems and get feedback and also have cookie parties and have fun.

I would like to thank my friend *Mahsa*, formerly from UPMC, who made my research visits to Paris more enjoyable. *Carolina, Julian* and *Alejandro* also deserve my special thanks for their hospitality during my internship in IMDEA Software Institute.

I would like to acknowledge The Scientific and Technical Research Council of Turkey (TÜBİTAK) for providing me financial support under the National Doctoral Funding Program (2211).

I take this opportunity to express my profound gratitude to my family for their endless support and confidence in me. I would like to express my heartily thanks to *Murat*, my spouse and best friend, for his morale support and sense of humor, encouragement and understanding during my PhD journey.

TABLE OF CONTENTS

List of Tables	xiv
List of Figures	xv
Chapter 1: Introduction	1
1.1 Asynchronous Programs	1
1.2 Motivation	5
1.3 Contributions	9
1.4 Outline of the thesis	10
Chapter 2: State of the Art in the Analysis of Asynchronous Programs	11
2.1 Concurrent Software Verification	11
2.2 Some Approaches for Concurrent Software Analysis	14
2.2.1 Sequentialization of Multithreaded Programs	15
2.2.2 Systematic Concurrency Exploration of Multithreaded Programs	17
2.2.3 Data Race Detection	18
2.2.4 Related Work on the Analysis of Event-driven Mobile Applications	19
Chapter 3: Exploiting Synchronization in the Analysis of Asynchronous Programs	22
3.1 Introduction	22
3.2 Shared Memory Asynchronous Programs	26
3.3 The DFW Scheduler	30
3.4 Compositional Semantics	35

3.5	Sequentialization	39
3.6	Complexity	43
3.7	Empirical Evaluation	47
3.8	Related Work	51

Chapter 4: Systematic Exploration of Asynchrony Bugs in Android

	Apps	54
4.1	Overview	54
4.2	AsyncDroid Design	55
4.2.1	AsyncDroid Workflow	55
4.2.2	Recording and Repeating User Events	57
4.2.3	Thread Scheduling	58
4.3	Case Studies	58
4.4	Limitations and Discussion	63
4.5	Related Work	63

Chapter 5: Characterization of the Correctness of Event Driven Asynchronous Programs: Robustness Against Asynchrony

5.1	Introduction	65
5.2	Motivating Robustness as Correctness Criteria	68
5.2.1	A Violation to Event Serializability	68
5.2.2	Violations to Event Determinism	70
5.2.3	A Robust Program	73
5.3	Event-Driven Asynchronous Programs	75
5.3.1	Program Syntax and Semantics	75
5.3.2	Weak Asynchronous Semantics	77
5.3.3	Strong Asynchronous Semantics	79
5.4	Robustness of Asynchronous Programs	81
5.4.1	Strong vs Weak Asynchronous Semantics	81

5.4.2	Event-Serializability and Event-Determinism	82
5.5	Conflict Robustness	83
5.5.1	Conflict-Event Serializability.	83
5.5.2	Conflict Determinism.	86
Chapter 6:	Verifying Robustness Against Asynchrony	88
6.1	Checking Conflict Determinism	90
6.1.1	Simulating Borderline Violations	93
6.1.2	Witnessing Borderline Violations	100
6.1.3	Reduction to the Procedural Semantics	101
6.2	Checking Conflict Robustness	104
6.2.1	Simulating Borderline Violations	106
6.2.2	Witnessing Borderline Violations	107
6.3	Experimental Evaluation	110
6.3.1	Checking Event-serializability of Execution Traces	111
6.3.2	Checking Robustness Against Asynchrony	115
6.4	Discussion of Special Cases	119
6.4.1	Waiting for an Asynchronous Task	120
6.4.2	Inter-related Events	121
6.5	Related Work	122
6.5.1	Robustness Against Asynchrony vs Data Race Freedom Violations	123
Chapter 7:	Conclusion	127
	Bibliography	129

LIST OF TABLES

4.1	Experimental data for AsyncDroid case studies	62
6.1	Experimental data for checking conflict-event serializability of traces.	115
6.2	Experimental data for conflict determinism.	116
6.3	Experimental data for checking conflict serializability.	118

LIST OF FIGURES

1.1	Asynchronous execution models	3
1.2	A problematic asynchronous C# code sample from MSDN	7
1.3	A code excerpt with an asynchrony bug from Ireccloud Android app	8
3.1	This code contains a subtle bug due to a race condition on the <code>m_bmp</code> field.	23
3.2	Transition rules over program configurations.	28
3.3	A tree of the $DF(K)$ scheduler	32
3.4	A program whose valuations are all reachable in $DFW(0)$, yet are not all reachable in $DF(K)$, for any $K \in \mathbb{N}$	34
3.5	The compositional program semantics.	36
3.6	Simulating an asynchronous program using compositional semantics.	40
3.7	The K -delay sequentialization $\Sigma(P, K)$	41
3.8	Translation to construct the verifier program	46
3.9	A preprocessing step for the $DF(K)$ sequentialization to remove wait statements.	48
3.10	Time to bug detection for some benchmarks	50
3.11	Time to bug detection for the parametrized example	50
4.1	The workflow of AsyncDroid	56
4.2	An asynchrony bug in a sample application	59
5.1	A program with event-serializability violation.	69
5.2	A program with event-determinism violation.	71
5.3	Two executions of an event leading to different final states	72

5.4	A robust program sample.	73
5.5	A canonical program syntax	76
5.6	The weak transition function.	78
5.7	The strong transition function.	81
5.8	Execution traces τ_1 and τ_2 and their conflict-event graphs.	84
5.9	An event e , an e -trace τ and the conflict-invocation graph of τ	87
6.1	Simulating borderline conflict determinism violations on the strong semantics.	92
6.2	Simulating a DFS-serial trace on the strong semantics.	95
6.3	Simulating borderline conflict serializability violations with a sequential program.	105

Chapter 1

INTRODUCTION

Asynchronous programming is widespread in modern software. Mobile, web and system applications are written asynchronous to be responsive to user requests and make use of multiple processing cores. The research in this dissertation aims at detecting asynchrony bugs and verifying the correctness of the shared memory asynchronous programs. It characterizes the correctness of the asynchronous programs and develops efficient algorithmic methods to verify the criteria.

In this section, we provide a brief explanation of asynchronous programs and how they run in different execution models. Then, we motivate the need for the concurrency analysis of the asynchronous programs by showing some real concurrency bugs from different platforms. Finally, we list our contributions for building bug detection tools and characterizing and verifying the correctness of asynchronous programs.

1.1 Asynchronous Programs

Asynchronous programs have a different control and data flow than the synchronous programs. They involve asynchronous procedure calls that are dispatched and executed asynchronously outside of the flow of the caller program. Different from a synchronous procedure call where the caller waits for the procedure to complete its work and return, an asynchronous procedure returns control to its caller before completing its job. This prevents the caller to be blocked during a long operation and enables it to continue processing other user requests.

Responsiveness is a hard requirement for event driven applications, so they heavily make use of asynchrony. They implement some event handler procedures to be

invoked in response to some user or system events. These event handlers make use of asynchronous procedures to offload some long operations to be executed in the background. Consider a simple web application. In response to a user request to display a web page, the application asynchronously downloads some images on the web page. During the download operation, it can process other user inputs. To run some operations after the completion of the asynchronous download operation, the application can register a callback that displays the downloaded image. The callback procedure is invoked asynchronously when the asynchronous procedure completes its job. This model of asynchronous programming prevents blocking and increases responsiveness.

Depending on the execution model, the asynchronously called procedures (during the dissertation will also be called as asynchronous tasks) can be executed concurrently and in parallel to the caller or operate on an isolated or shared memory. This dissertation focuses on shared memory asynchronous programs.

In the single threaded execution model, asynchronously called procedures, callbacks and the handlers of some user and system events are collected in a task buffer. An application level dispatcher dispatches a task from the task buffer and runs it to completion, one after the other. The tasks can be dispatched in FIFO (first-in-first-out) or another order depending on the system specifications. JavaScript is an example of a single threaded asynchronous system. It has a single thread executing the user events and the callbacks one after the other, serially. This model is also called the event loop model based on its implementation as a loop processing the messages in the message queue.

Figure 1.1.(a) illustrates this execution model. The asynchronous tasks to run procedures A and B run serial to each other, probably with any order, after the completion of main method. The execution of these procedures does not interleave. However, there is some concurrency in the execution of these procedures: (1) due to the nondeterminism in the execution order of A and B , (2) due to the nondeterminism in the execution of their callbacks. If the procedures A and B register some callbacks

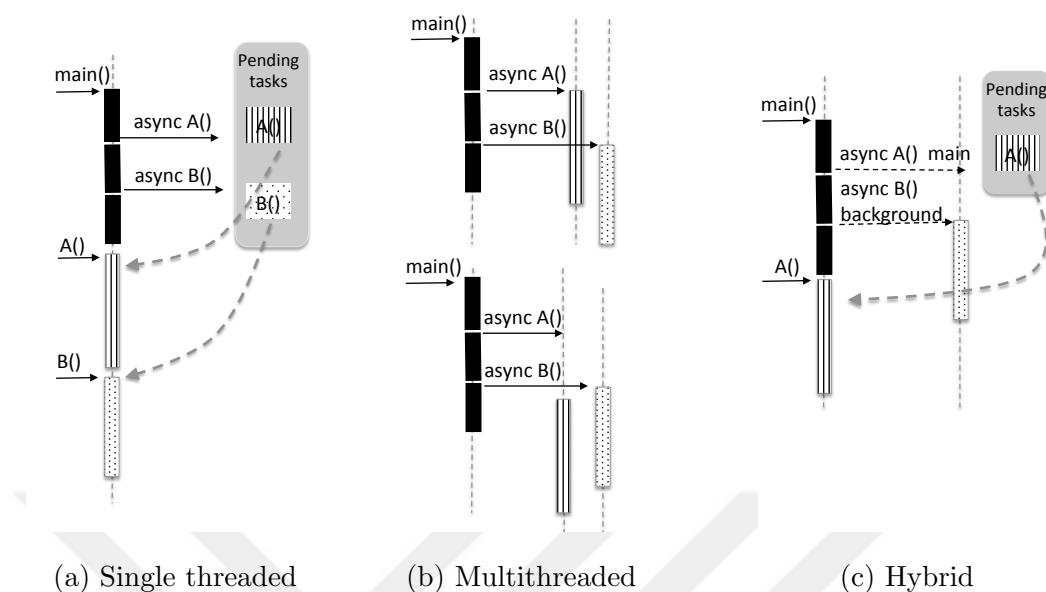


Figure 1.1: Asynchronous execution models

on their completion (e.g. to be run when the network response is received), the order of their insertion into and dispatch from the task buffer will be nondeterministic.

As the use of multicore devices gets widespread, some programming models involve parallelism for better performance. They run all asynchronous procedures potentially in parallel in a multithreaded fashion. Such programs might require the use of some synchronization primitives (e.g. protected code segments for mutual exclusion, waiting for a result of an asynchronously called procedure, etc).

The asynchronous procedures *A* and *B* (Figure 1.1.(b)) run in parallel and their execution may interleave. C# asynchronous task implementation in Task Parallel Library (TPL) is an example execution model where tasks can run in parallel (the degree of parallelism depends on the thread pool size of the application). We can also consider dynamically created threads as a form of asynchronous task invocation, as the caller of the task does not wait for the operation running in the thread to complete. Examples of such multithreading libraries are present in various languages and frameworks.

Hybrid execution models involve both event looping and thread parallelism. Especially event-driven asynchronous programs on mobile devices utilize this model for better responsiveness. They involve a main thread that performs operations guaranteed to complete quickly (e.g. UI component updates) and delegate any possibly long-running operation into an asynchronous task running on a background thread. Android asynchrony is an example for the hybrid model. In this model, the background threads can have task buffers as well. A thread can post an asynchronous task to its own or another thread's task buffer. While this model utilizes multithreading, it also provides guarantee for serial execution of the asynchronous tasks that are posted to the same thread.

Figure 1.1.(c) illustrates such an asynchronous execution model. The asynchronous tasks invoked on the same thread (i.e. the *main* and *A* procedures in this example) run serial to each other and their execution do not interleave. The execution of the procedures invoked on different threads (i.e. the execution of the procedure *B* may interleave with the rest of the *main* procedure after invoking procedure *B* and the procedure *A*.

In some systems, threads do not communicate over a shared memory but by asynchronous message passing, i.e. by only sending asynchronous tasks or messages to each other's task buffers. Asynchronous message passing is especially used for writing reliable programs for distributed systems. Single-process JavaScript programs or multi-process Erlang or Scala programs are examples for such message passing programs. In this dissertation, we consider asynchronous programs with possible thread interference, where the threads access and interleavingly operate on a shared memory.

Programming languages provide some asynchronous programming models so that the programmers can easily write asynchronous programs. JavaScript's asynchronous library methods, Java *futures*, C# *Tasks*, *async/await* model and the Android library's asynchrony constructs (*AsyncTask*, *Handler*, etc) are some examples for the programming language implementations for some asynchronous programming models.

In this dissertation, the correctness of an asynchronous program is characterized independent of the computing platforms. Modern computing platforms (e.g. JavaScript execution engines or mobile phones) and their programming models change frequently, in subtle ways. For instance, the Android asynchrony construct `AsyncTask` uses a single background thread and hence guarantees the serial execution of `AsyncTask` asynchronous tasks in Android versions after 1.6 and before 3.0, while multiple background threads are used in the other versions. JavaScript is currently single-threaded, but there is ongoing work on multithreaded JavaScript execution engines.

Our work in Chapter 3 focuses on the asynchronous programs whose asynchronous invocations run potentially in parallel to each other (e.g. as in C# programs). In Chapter 4, we work on Android applications which have a hybrid execution model with both serial and parallel execution of asynchronous invocations.

In Chapter 5, we characterize a correctness criterion for the asynchronous programs independent of the programming language and execution platform. We provide abstractions for the event-driven shared memory asynchronous programs by abstracting away language and platform details. We characterize correctness based on two abstract views of the system: a restricted high-level view and a concurrent low-level view. The high-level view is a reference model with a single thread handling user events in a serial manner, one after the completion of all asynchronous tasks invoked by a previous event handler. The low-level view involves the concurrency of the actual program in which the asynchronous tasks running on multiple threads interact over shared memory.

1.2 Motivation

Asynchronous programs are prone to concurrency errors and there is a need for an analysis to detect the concurrency bugs in these programs. In this chapter, we motivate this need by presenting real concurrency bugs in the asynchronous programs from different platforms. Then, we argue that the existing tools and techniques for the analysis for classical multithreaded programs with a fixed number of threads are not

directly applicable or well-suited to the asynchronous programs which dynamically create an unbounded number of asynchronous tasks.

Due to the asynchrony and the interference on the shared memory, writing correct asynchronous programs is hard. Asynchronously invoked procedures induce different data and control flow than the synchronous procedure calls. Different executions of an asynchronous program are possible depending on the timing of the asynchronous procedures. In addition to the difficulties posed by the asynchronous execution model, the programs running multiple threads inherit the generic difficulties of multithreaded programs. The threads interleavingly access the shared memory and depending on the execution, different valuations of the program variables might be possible.

Although asynchronous programming models aim to ease asynchronous programming and provide some programming language constructs, writing correct programs is still challenging. The asynchrony constructs obscure the data and control flow of a program in different ways and they introduce subtle interleavings. The programmers do not consider all possible interleavings while reasoning about their programs. Many buggy asynchronous code can be found in real programs and applications provided in the open repositories and discussion forums.

The following examples, simplified from real programs, motivate the need for such an abstraction for the asynchronous event driven programs. They show that even in the apparently simple asynchronous programs, it is easy to neglect some subtle interleavings and write programs that may exhibit some unexpected program behaviors.

Figure 1.2 shows a simplified version of an asynchronous program from MSDN [Cod,]. Language specific asynchrony constructs are omitted for the sake of simplicity. Briefly, the `Button1_Click` procedure calls the `SendData` method to create a web request and set the response into `m_GetResponse` variable. `SendData` is an asynchronous method and it returns to its caller before completing its job, i.e. before setting `m_GetResponse`. The programmer expects `m_GetResponse` to be set within two seconds after calling the `SendData` method. However, the execution of the asynchronous

```
String m_GetResponse;

void Button_Click(...) {
    SendData(url) // an async procedure
    ... // code to wait for two seconds
    ... // print m_GetResponse
}

void SendData (String url) {
    // send web request
    // get response
    m_GetResponse = ...
}
```

Figure 1.2: A code excerpt simplified from an asynchronous C# code from MSDN. This code contains a subtle bug due to the racy accesses to the `m_GetResponse` field.

tasks may not satisfy the programmer assumptions about the timing of the tasks. Depending on the interleavings in the execution, the program can work as expected or `m_GetResponse` may not be set and the program can hit a null pointer exception.

One way to prevent the problem may be to modify `SendData` to return a handle of the asynchronous task so that the caller waits for the completion of this task before accessing `m_GetResponse`. Another solution might avoid blocking the caller. A callback procedure might be defined in the code that will be invoked after the completion of `SendData`. It is a common rule of thumb to avoid blocking or running long-running jobs in the main (UI) thread that processes the user inputs.

Android library provides `AsyncTask` convenience class along with some other constructs, to offload long running jobs to background threads and asynchronously use the produced results in the main thread, without blocking it. Despite the apparent simplicity of applying rules of thumb, subtle executions due to asynchrony can result in unexpected application behaviors. Figure 1.3 shows a sample code from a version of the IrcCloud [Irc, a] app that uses `AsyncTask` to handle sending data in the background.

Different from MSDN example that can produce different behaviors for the event handler procedure `ButtonClick`, processing of more than two user inputs are involved in the problematic execution of this program. The programmers tend to expect an

```
EditText msgTxt;

// Event handler 1: Runs when the user clicks on the send button
boolean onKey(...) {
    if(...) // check connection, etc )
        // asynchronously calls doInBackground() to send message
    }

void doInBackground () { // runs in a background thread
    send msgTxt.getText()
}

// Event handler 2: Runs when the user clicks on the username
void onUserDoubleClicked(String name) {
    ...
    text += " " + name;
    msgTxt.setText(text);
}
```

Figure 1.3: A code excerpt with an asynchrony bug from Irecloud Android app

event handler to run in isolation from the other event handlers, however the execution of more than one event handlers interfere with each other due to the asynchrony and concurrency. In the example scenario, the user first clicks on the send button to send the message contents to the web, and then double clicks on the username displayed on the screen. Depending on the interleaving of the second event handler running on the main thread and the `doInBackground` method running on asynchronously on the background thread, a wrong content of the message (appended with the username) may be sent to the network. The program can be corrected by copying the current content of the message text and sending this copy in the `doInBackground` method. This solution ensures the program to run as intended regardless of their interleaving execution on the shared memory.

The techniques and tools for the analysis of sequential and classical multithreaded programs are not sufficient for the

The goal of this dissertation is to characterize the correctness of the event-driven asynchronous programs that (i) provides an abstraction to meet programmer expectations (ii) captures the concurrency bugs found in the programs as violations and (iii) is independent of the application logic or the underlying execution platform. Currently, there are no such mature programming abstractions or correctness criteria

to satisfy these goals.

1.3 Contributions

This thesis makes the following contributions:

Building methods and tools for detecting asynchrony bugs:

- Developing a static, sequentialization based method to detect the asynchrony bugs in the shared memory asynchronous programs. Our contribution is to exploit the synchronization constructs in the program to reduce the cost of the analysis. The improvement of our technique is significant when the program has statements to wait for some other asynchronous tasks for their completion.
- Adapting dynamic concurrency exploration technique to event driven programs. We build the first concurrency testing tool to detect the asynchrony bugs in Android applications.

Characterizing and verifying the correctness of asynchronous programs:

- Characterizing the correctness of the asynchronous event driven programs as *robustness against asynchrony* that captures the programmer assumptions about the asynchronous program execution. The proposed correctness criteria is generic and independent of the program logic.
- Showing that *robustness against asynchrony* can be formulated as a conjunction of two properties: *event determinism* and *event serializability*.
- Developing efficient algorithmic methods to check whether an asynchronous program is robust, i.e. event deterministic and event serializable. Our method reduces the analysis of an event-handler asynchronous program with an unbounded number of asynchronous task invocations to a sequential program analysis. When

the data domain is bounded, checking conflict-robustness for event-driven asynchronous programs (with recursive procedure calls and unbounded number of events and asynchronous tasks) is polynomial-time decidable.

1.4 Outline of the thesis

This dissertation is organized as follows: Chapter 2 summarizes the state of the art in the verification of asynchronous and concurrent programs. Chapter 3 presents our sequentialization-based method for detecting bugs in the asynchronous programs. Chapter 4 presents AsyncDroid dynamic concurrency exploration tool for Android applications. The main contribution of the thesis is presented in Chapters 5 and 6. Chapter 5 presents our characterization of the correctness of asynchronous programs, “robustness against asynchrony”. The algorithmic methods to verify robustness and the experimental work on checking robustness of Android applications are provided in Chapter 6. Chapter 7 concludes the thesis by summarizing the features of our proposed correctness criterion and some possible directions for the future work.

The first two contributions have been published in [Emmi et al., 2014] and [Ozkan et al., 2015] respectively. The work about robustness against asynchrony is currently under submission.

Chapter 2

STATE OF THE ART IN THE ANALYSIS OF ASYNCHRONOUS PROGRAMS

As the nature of the event-driven asynchronous programs is different than the classical multithreaded programs, the existing software analysis tools for shared memory multithreaded programs are generally not well-suited to verifying event-driven asynchronous programs. Recent work on the analysis of these programs either perform race detection that identifies low-level, generic errors that are symptomatic of higher-level programming errors or apply dynamic exploration techniques which only alarm in case of an application crash or programmer specified assertion violation.

Our goal in this dissertation is to characterize a correctness criteria for event-driven asynchronous programs and build concurrency analysis tools specific for asynchronous programs. In this chapter, we briefly provide some background information on the software verification and the bug detection or verification approaches close to the work in this dissertation. The discussion of related work with a comparison to our contributions will be provided in each chapter, in Sections 3.8, 4.5 and 6.5.

2.1 Concurrent Software Verification

Software verification is a fundamental and computationally hard problem that roots back to the work of Turing. In this dissertation, we work on the verification of *safety* properties. Safety properties state the program constraints which must not be violated and safety verification guarantees bad things do not happen during the program computation. In broad sense, a safety verification problem takes a program together with some assertion statements e.g. some variable is not null, some value is greater than zero, etc. The safety verification process checks whether the assertions in the

program are valid.

Deductive verification, *model checking* and *abstract interpretation* are some popular approaches for tackling the problem of program verification. Deductive verification (aka theorem proving) states the correctness of the program as a set of mathematical formulas that are solved by theorem provers. Model checking, models the program behavior as a transition system and checks some properties by analyzing state reachability in the transition system. Abstract interpretation performs reasoning on an approximation of the program semantics based on some mappings between concrete and abstract states of the program. While these approaches have inherent advantages and disadvantages, modern techniques and tools in software verification utilize a combination of these approaches [Jhala and Majumdar, 2009], e.g. model checking symbolic abstractions of program states, using formula satisfiability methods for model checking software, etc. The software analysis approaches in this dissertation are not limited to but closer to the model checking based approach.

Complexity of safety verification. The problem of assertion verification can be reduced to the problem of state reachability, i.e. whether it is possible to reach a program state where a particular variable has a particular data value. We consider programs with finite data where the program variables can get values over a finite domain (e.g. booleans, bounded integer values).

The complexity of concurrent program analysis is dependent on the features of the concurrent program model. A finite-data program without recursive procedure calls can be modeled using a finite state system. The reachability in the finite-data programs with finite number of threads is known to be PSPACE-complete [Kozen, 1977]. The analysis complexity increases as the expressiveness of the program increases.

Synchronous Procedure calls. A program with recursive synchronous procedure calls can be modeled using a stack to keep the procedure invocations. Then, the program state reachability reduces to the reachability in concurrent systems with stacks. The verification problem for the multithreaded programs with recursive procedure calls is undecidable even for a fixed number of threads [Ramalingam, 2000].

Asynchronous Procedure calls. Asynchronous procedure calls are not invoked immediately as synchronous calls but they are collected in a task buffer and invoked at a later time out of the caller procedure's control flow. Depending on the execution model, the asynchronous procedure calls in the task buffer can be invoked serially on the same thread or they can be invoked on different threads and run in parallel in multithreaded fashion.

Semantics of the asynchronous task buffer. The semantics of the task buffer varies for different execution models. A popular approach is to run the asynchronous tasks in a task buffer serially, dispatching an asynchronous task only after the completion of the running task. Programs maintaining first-in-first-out (FIFO) task buffers leads to undecidable state reachability as the reachability in the finite state systems with FIFO queues is undecidable. Programs with unordered task buffers (also called as bags) dispatches the asynchronous tasks in a nondeterministic order. Theoretically, unordered tasks allows for multiset and counting abstractions and yields decidable reachability. The works by [Sen and Viswanathan, 2006, Jhala and Majumdar, 2007] analyze asynchronous program verification with a focus on the serial unordered task buffer model. The state reachability for such programs is shown to be EXPSPACE-complete [Ganty and Majumdar, 2012]. For many subclasses of programs where the asynchronous tasks are executed in parallel in an isolated, interleaving-free fashion, the complexity of the analysis is shown to be decidable [Bouajjani and Emmi, 2012a].

In this dissertation, we target the asynchronous program model which involves the parallel execution of asynchronous tasks on multiple threads that interleavingly operate on a shared memory, falling into the undecidable case. Different from the multithreaded programs with a fixed number of threads, these programs have unbounded number of tasks that may run in parallel.

Complexity of Bounded Verification. To overcome the undecidability of the verification, the work in the literature either explore an abstracted overapproximate semantics or underapproximate the program behavior by considering a subset of possible interleavings. The latter approach parametrizes and prioritizes the search space

with a limited set of program behaviors that is defined by the bounding parameter. This approach allows for iterative systematic exploration by iteratively increasing the bounding parameter to cover more program behaviors. One approach for underapproximated analysis is depth-bounding [Russell and Norvig, 1995, Udupa et al., 2011]. Depth bounding limits the executions under analysis with a bounded number of program steps. Though it is applicable to some programs, it is inadequate to capture the interesting behavior for multithreaded programs. An effective approach is context-bounded exploration [Qadeer and Rehof, 2005] [Musuvathi and Qadeer, 2007]. Context bounding, bounds the number of context switches (i.e. switching the execution from one thread to another) in the execution of a multithreaded program. The k -context bounded exploration of a program captures all possible executions of the program where at most k context switches are made. The fact that most concurrency bugs are exposed in a small number of context switches makes context bounded analysis suitable for finding bugs in concurrent programs. Although context bounded computations still have an infinite state space due to the recursive procedure calls, its analysis is shown to be decidable for multithreaded programs [Qadeer and Rehof, 2005]. The work by [Atig et al., 2009] considers a variant of context bounding for the analysis of programs with unbounded number of threads. Their approach does not bound the total number of context switches but bounds the context for each thread and yields decidable state reachability.

2.2 Some Approaches for Concurrent Software Analysis

In this section, we provide the techniques for the analysis of concurrent software which are close to the work presented in this dissertation. In Chapter 3, we work on a sequentialization based technique for the verification of asynchronous programs. Our work in Chapter 4 is an adaptation of systematic concurrency exploration of multithreaded programs to event-driven asynchronous programs. In Chapter 5, we present a correctness criterion for these programs, which aims to provide a high-level correctness criterion different from the data races which is widely studied for the

detection of concurrency bugs. In this section we provide a brief summary for data race checking to make the distinction with our correctness criterion “robustness against asynchrony” more clear.

2.2.1 Sequentialization of Multithreaded Programs

One of the advantages of context-bounded analysis is that the executions characterized by k context bound can be encoded as a sequential program. The sequentialized program represents the set of k -context bounded executions of the original multithreaded program. Then, the errors exposed in the concurrent execution of the program can be detected by performing a sequential program analysis on the sequentialized program. The approach has become popular as it reduces the multithreaded program analysis into a sequential program analysis and hence can leverage the existing program analysis tools. The KISS verifier [Qadeer and Wu, 2004] pioneered this approach by underapproximating the program behaviors into a sequential program by providing a transformation of the source code from multithreaded programs into sequential programs.

The novel idea of sequentializing by nondeterministically guessing future values of the global variables and later constraining them with the appropriate values is introduced by [Lal and Reps, 2009]. Their sequentialization of the program replaces the control nondeterminism due to the thread interleavings in the concurrent programs by data nondeterminism. The valuations of the variables are nondeterministically guessed and symbolically represented. At a later point in the program, the symbolic data valuations are constrained so that they form valid valuations of the concurrent executions of the original program. The work by [Lal and Reps, 2009] presents a source to source translation of round robin executions that captures a context-bounded underapproximation of a program for any context-bound. The work by [LaTorre et al., 2009] provides another approach to the sequentialization which avoids the exploration of the states unreachable in the original concurrent program. The sequentialization work by [LaTorre et al., 2010, Torre et al., 2012] handles the analysis of unbounded number

of context switches.

The complexity of context-bounded analysis is exponential in the number of context bound. For an efficient analysis of a multithreaded program, the context bound must be in the order of the number of threads to switch the context and give change to the execution of each task. Since asynchronous programs may create an unbounded number of tasks that are potentially invoked on different threads, and the analysis becomes exponential in the number of threads. Delay bounding [Emmi et al., 2011a] provides another characterization of underapproximated analysis of concurrent programs that is effective for the analysis of unbounded number of threads. It does not bound the number of context switches between the tasks, but bounds the number of deviations from a deterministic schedule of tasks. The k -delay bounded set of program behaviors is defined on a default deterministic task scheduler that has a nondeterministic choice to “delay” the execution of a scheduled task. In a k -delay bounded exploration of a program, the scheduler can delay the execution of a scheduled task only for k times. The delayed task is scheduled back after scheduling the other tasks in the buffer at the time of its delay. This characterization of exploration is efficient for asynchronous programs since the analysis is exponential in the number of delays and the number of delays is independent from the number of tasks in an execution.

Phase bounding [Bouajjani and Emmi, 2014] is proposed for asynchronous programs running on multiprocessors whose processes communicate by posting tasks/messages to each other’s processors. Though it bounds the number of process communication cycles and explores a bounded portion of the state space, the reachability problem for phase bounded multiprocessor programs is undecidable. Phase bounding is used in combination with delay bounding for algorithmic purposes [Bouajjani and Emmi, 2014].

Our sequentialization work on shared memory asynchronous programs in Chapter 3 is built on top the delay bounded analysis in [Emmi et al., 2011a] which can handle asynchronous programs with unbounded number of tasks. We define our sequentialization using a general compositional framework for sequentializations presented in

[Bouajjani et al., 2011]. The framework summarizes the effect of each asynchronous task into a bounded task interface mainly composed of the symbolic valuations of the global variables. Then, it generalizes the sequentialization schema as a tree traversal of these task interfaces. In our work, we show that we can reduce the exploration cost by exploring the deviations from a scheduler which takes the program synchronization (e.g. waiting for the completion of a task) into account.

2.2.2 Systematic Concurrency Exploration of Multithreaded Programs

Systematic concurrency exploration tools focus on the exploration of nondeterminism due to scheduling choices. Typically, these tools examine different executions of a program for a given sequence of user interactions. They systematically traverse the set of possible schedules and execute the program enforcing each schedule. During the execution, they dynamically check whether the assertions are satisfied in a program. Java PathFinder [JPF,] model checker for Java programs, Cmc model checker [Musuvathi and Engler, 2004] for C programs and Chess [Musuvathi et al., 2007] for multithreaded Windows programs are some well known concurrency exploration tools. Chess applies the aforementioned iterative context bounding [Musuvathi and Qadeer, 2007] and it is also adapted for delay bounded exploration in [Emmi et al., 2011a]. Bbs tool [Majumdar and Wang, 2015] implements phase-bounding proposed in the work by [Bouajjani and Emmi, 2014] for the analysis of embedded C programs.

Our work on systematic exploration for detecting bugs in Android applications in Chapter 4 adopts a delay bounding approach to event-driven programs. We explore different possible concurrent behaviors for a given, fixed user interaction by recording a sequence of inputs and replaying them over a set of schedules.

An independent work [Desai et al., 2015] applies delay bounding for systematic testing of asynchronous P programs. To improve scalability, they explore different interleavings of a program by iteratively increasing the delay bound. The work combines the systematic delay bounding search with a randomized approach. Beyond a certain delay bound reached after a time budget, they exercise random schedules to

cover some executions with higher delay bounds.

Recent work [Maiya et al., 2016] proposes a partial order reduction technique to reduce the state space of executions to be explored, for asynchronous programs with multiple threads with message queues. It builds and tracks a dependence relation suitable for event driven programs and examines different orderings of the events only if necessary. The systematic concurrency exploration tools for event driven programs can be improved to employ this partial order reduction technique and explore fewer number of executions.

2.2.3 Data Race Detection

A data race occurs when two threads access a data variable concurrently, and at least one of these accesses is a write [Netzer and Miller, 1992]. Data races are often considered as a sign of concurrency bugs, since the outcome of the programs with data races may depend on the order of the memory accesses and some executions can lead to unintended program behavior. Data race detection is a popular approach for the detection of bugs in the multithreaded programs [Savage et al., 1997, Elmas et al., 2007, Flanagan and Freund, 2009] and recently applied to event driven web and mobile applications [Petrov et al., 2012, Raychev et al., 2013, Mutlu et al., 2015, Maiya et al., 2014, Hsiao et al., 2014, Bielik et al., 2015].

A program can be checked for data races at runtime (aka dynamic data race detection) or at compile time by analyzing the source program without actually running it (aka static data race detection). Most state-of-the-art data race checking tools are dynamic. Lockset-based analysis and happens-before analysis are two well-known approaches for the dynamic detection of data races. Lock-set based detection searches for the potential races due to the accesses to a shared memory location that is not protected by common locks. Happens-before analysis uses a happens-before relation that represents the causal order between the accesses to the memory locations. The race checking analysis detects the accesses to a location without any causal ordering constraints.

Data races are often a low-level symptom of higher-level concurrency bugs. It is likely that a detected data race do not actually cause a buggy program behavior. As an example, consider an execution of an application where an event handler writes to an object to assign an object to a null pointer and a concurrent read access by another event handler checks whether the pointer is null before using it. Such a program may be bug free despite the racy memory accesses by two event handlers. The correctness criteria we propose in this dissertation is not violated by these racy accesses if there are no cyclic dependencies between the two event handlers.

In this dissertation, we aim to characterize a higher level correctness criteria which abstracts programmer expectations about the timing of the asynchronous execution and the interference between the parts of the program. Though data race freedom violations and the violations to our correctness criterion “robustness against asynchrony” are not comparable, we provide a brief discussion of the bugs and false positives that can be found using data race checking and robustness checking in Section 6.5.1.

Another advantage of robustness checking over data race checking is its computational complexity. Data race detection is an NP-complete problem in general [Netzer and Miller, 1990]. The dynamic race checking tools collect a set of execution traces and check for the data races on the collected races. Though they apply some heuristics to collect diverse execution traces and detect as many races as possible, they do not guarantee detecting all possible races in a program. We present an efficient method that *verifies* a syntactic notion of robustness against asynchrony, *conflict robustness*, given a program with some user inputs. Surprisingly, our method reduces the analysis of an asynchronous program with unbounded number of asynchronous tasks into a sequential program analysis and yields a polynomial complexity.

2.2.4 Related Work on the Analysis of Event-driven Mobile Applications

Recently, the Android platform has been widely studied in the context of software analysis.

Many of the existing approaches to bug detection targets the user interface bugs and

the nondeterminism due to user interaction. They test an application for different user interface inputs [Hu and Neamtiu, 2011, Anand et al., 2012a, Azim and Neamtiu, 2013a, Choi et al., 2013, Machiry et al., 2013a] with an aim to cover a large set of application behaviors. Exploration of input nondeterminism and concurrency nondeterminism are complementary to each other. Running the application with some particular user inputs does not test all behaviors of the application for these inputs. Because, more than one possible execution of the same user inputs is possible due to the concurrency nondeterminism.

In this dissertation, we focus on concurrency nondeterminism and explore different behaviors of an application for a fixed user input sequence. For the concurrency analysis of Android applications, race detection has been the focus of much work [Maiya et al., 2014, Hsiao et al., 2014, Bielik et al., 2015]. There is not a reference formal definition of the happens-before relation for the Android platform. These works define the happens before order based on some subset of the ordering guarantees in the platform. As a result, they produce high number of false positives and even runtime techniques that precisely check for races require manual inspection of racy executions afterwards. A detailed comparison of race detection and checking for our correctness criterion is provided in Section 6.5.1.

The work in [Lin et al., 2014] analyzes the use, misuse, and underuse of asynchrony in Android applications and refactors them so that they utilize asynchrony to achieve better performance. They provide a program translation to move long running jobs to asynchronous tasks. They ensure that the refactored asynchronous tasks do not race with their callers while refactoring the program. In [Lin et al., 2015], they extend their work to transform existing improperly-used asynchrony constructs into a proper usage is provided in Android programming model.

Recent work [Safi et al., 2015] presents a static analysis technique to detect event anomalies in the event driven programs. An event anomaly arises due to more than one event handlers which access the same memory location and can be triggered in a nondeterministic order. A typical program has many event handlers with common

memory location accesses and the work finds high number of event anomalies in the analyzed applications. While this definition of event anomalies and the presented technique can be used to uncover bugs, it is likely that the detected event anomalies do not cause buggy behavior.

Overall evaluation of the existing work for the analysis of asynchronous programs. Although the detection of concurrency bugs in asynchronous and event driven shared memory programs is getting more popular, there is not a formalization for a generic correctness criteria (e.g. serializability for database transactions, linearizability of concurrent data types) for these programs. In this dissertation, besides developing methods for the analysis of asynchronous programs, we propose a generic, high level correctness criteria, *robustness against asynchrony*.

Chapter 3

EXPLOITING SYNCHRONIZATION IN THE ANALYSIS OF ASYNCHRONOUS PROGRAMS ¹

This chapter presents our program analysis method to detect program errors in the form of assertion violations in the shared memory asynchronous programs. Our method follows the line of prioritized exploration research which aims at exploring a limited set of program behaviors that are likely to expose programming errors. We find that the limited exploration of asynchronous program behaviors can be made more efficient by designing parameterized schedulers which better match the inherent ordering of program statements, e.g., arising from waiting for an asynchronous task to complete. We follow a *sequentialization* approach to analyzing asynchronous programs, which leverages existing sequential program analysis tools by encoding asynchronous program executions as the executions of a sequential program.

3.1 Introduction

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which “asynchronous procedures” can execute concurrently with their callers, until their callers explicitly wait for their completion. Accordingly, as concurrently-executing procedures interleave their accesses to shared memory, asynchronous programs are prone to concurrency-related errors.

Consider the subtle error in the event-handling C# code of a graphical user interface found on StackOverflow, which is listed Figure 3.1. The `MySubClass.OnNavigatedTo`

¹The work in this chapter is a joint work with Michael Emmi and Serdar Tasiran. [Emmi et al., 2014]

```
// MySubClass
BitmapImage m_bmp;
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    await PlayIntroSoundAsync();
    image1.Source = m_bmp;
    Canvas.SetLeft(image1, Window.Current.Bounds.Width - m_bmp.PixelWidth);
}
protected override async void LoadState(Object nav, Dictionary<String, Object> pageState)
{
    m_bmp = new BitmapImage();
    var file = await StorageFile.GetFileFromApplicationUriAsync("ms-appx:///pic.png");
    using (var stream = await file.OpenReadAsync())
        await m_bmp.SetSourceAsync(stream);
}
// base class
class LayoutAwarePage : Page
{
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        // ...
        this.LoadState(e.Parameter, null);
    }
}
```

Figure 3.1: This code contains a subtle bug due to a race condition on the `m_bmp` field.

method accesses image-related information (`m_bmp.PixelWidth`) which is filled in by the `LoadState` method, invoked by the `OnNavigatedTo` method of the base class. However, the `LoadState` method has been implemented to execute asynchronously so that its callers can continue to execute while the image file is read — which is presumably a high-latency operation — meaning that `base.OnNavigatedTo` can return before `m_bmp` has been initialized. This creates a race between the initialization of `m_bmp` and its use in the call to `Canvas.SetLeft`, which results in an error when its use wins. Not having anticipated this race, the programmer has failed to provide adequate synchronization to ensure that the call to `LoadState` completes before `m_bmp` is accessed by `OnNavigatedTo`.

While detecting such concurrency bugs by *exhaustive* exploration of all possible program schedules is intractable, one promising approach is the *prioritized* exploration of behaviors whose manifestations rely on a small numbering of ordering dependencies between program operations. In particular, the delay bounding approach [Emmi et al., 2011a] explores the program behaviors arising in executions with a given scheduler $S(K)$ parameterized by a “delay bound” $K \in \mathbb{N}$; while $S(0)$ is a deterministic scheduler, exhibiting only one order of program operations, $S(K)$ is given additional nondeterministic choice with each increasing value of K , allowing additional orders, and ultimately, exhibiting additional observable program behaviors. The approach is particularly compelling under the hypothesis that interesting program behaviors manifest with few ordering dependencies: Emmi et al. [Emmi et al., 2011a] demonstrate an efficiently-implementable “depth-first” delaying scheduler $DF(K)$ which can expose behaviors with few ordering dependencies using small values of K .

In practice, the cost of prioritized exploration with a parameterized scheduler $S(K)$ is highly sensitive to the value of K , limiting $DF(K)$ -based exploration to roughly $0 \leq K < 5$, depending on program size. While such small values of K may suffice to expose bugs in programs which use very little synchronization, each program synchronization statement induces another event-order dependency, possibly forcing $DF(K)$ to further deviate from its natural deterministic order by increasing K . For instance, if $DF(K)$ ’s default schedule encounters a statement which acquires a lock held by another thread, then $DF(K)$ must spend one of its K delays in order to execute the other thread and eventually progress past the lock acquisition. In the context of asynchronous programs, e.g., using C#’s asynchronous methods, $DF(K)$ must spend one if its K delays to advance past a statement which waits for a non-completed task to complete. It follows that program behaviors which can appear only after a high number of synchronization statements carry a high number of event-order dependencies, which ultimately may be exercised by $DF(K)$ only for large values of K . As the cost of program exploration with $DF(K)$ is sensitive to K , the discovery of such behaviors may require an unreasonable amount of computing resources.

In this work, we demonstrate a delaying scheduler $\text{DFW}(K)$ for which the cost of exploration is not tied to program synchronization. $\text{DFW}(K)$ scheduler has the advantages of:

- **Sequentialization** The program executions allowed by $\text{DFW}(K)$ can be simulated by a sequential program with nondeterministically-chosen data values.
- **Low Complexity** The reachability problem for finite-data programs restricted to $\text{DFW}(K)$ executions is NP-complete in K^2 .

Analysis based on our new scheduler comes at no greater computational cost, and provides strictly greater behavioral coverage than analysis based on existing parameterized schedulers; we validate these claims both conceptually, with complexity and behavioral-inclusion arguments, and empirically, by discovering actual reported bugs faster with smaller parameter values.

The contributions of this work are:

- A formal semantics of asynchronous programs with synchronization.
- A formal description of the $\text{DFW}(K)$ scheduler, and comparison with the existing scheduler $\text{DF}(K)$ that is known to be efficient for asynchronous programs.
- A “compositional semantics” for $\text{DFW}(K)$, which fosters sequentialization.
- A code translation encoding $\text{DFW}(K)$ executions as a sequential program.
- NP-completeness of reachability under $\text{DFW}(K)$ for finite data programs.
- An empirical comparison: $\text{DFW}(K)$ finds bugs faster, with smaller K .

While our development is centered around a simple programming model with asynchronous procedure calls, and “wait” statements which block until the completion of a given asynchronous call, our technical innovations also apply to other asynchronous programming primitives provided by widely-used programming languages, such as

²This complexity assumes program variables are fixed in number and size, as usual [Lal and Reps, 2009, Emmi et al., 2011a].

the partially-synchronous procedure calls of C#³ and the wait-for-all synchronization barriers of, e.g., Cilk and X10. We believe that the same principles would also apply for other synchronization mechanisms such as semaphores and locks.

3.2 Shared Memory Asynchronous Programs

In order to develop our theory around synchronization-exploiting schedulers, we introduce a formal model of asynchronous programs with asynchronously executing procedure calls, and blocking wait-for-completion synchronization. When a procedure is called asynchronously, control returns immediately to the caller, who may store a *task identifier* with which to refer to the procedure instance, which we henceforth refer to as a *task*. The newly-created task then executes concurrently with the caller, possibly accessing the same set of global program variables concurrently. While we suppose for simplicity that task identifiers are not stored in global program variables, we do allow task identifiers to be stored in procedure-local variables, passed as arguments to called procedures, and returned from procedure calls. A task identifier i may be used to block the execution of another task j until task i completes, at which point the task's result may be stored in a program variable. Together these features comprise an expressive model of concurrent programs which corresponds closely to the features in a diverse range of programming languages including C#, Cilk, and X10.

Syntactically, a program is a set of global variable declarations, along with a set of procedure declarations whose statements are given by the grammar:

```

s ::= s ; s | assume e | skip | if e then s else s
    | while e do s | x := e | call x := p e | return e
    | async x := p e | x := wait e

```

Here, x ranges over the set **Vars** of program variables, p ranges over procedure names, and e ranges over program expressions — whose grammar we leave unspecified. The

³In C#, executing an “await” inside of a procedure returns control to the caller, executing the remaining continuation asynchronously.

set of program values **Vals** includes the set **IDs** of task identifiers, including a special polymorphic nil value $\perp$. We assume program expressions are statically typed, that each task-identifier typed expression evaluates to a single value $i \in \text{IDs}$, and that each non-identifier typed expression evaluates to a set of values $V \subseteq (\text{Vals} \setminus \text{IDs})$. Furthermore, we suppose that the set of program expressions contains $\star$, which can evaluate to any non-identifier program value, and that each program contains a single entry-point procedure named **main**.

Aside from the usual sequential programming statements, we include the statement **async** $x := p \ e$ which creates a new task to execute procedure p with argument e , storing its identifier in the procedure-local variable x , and the statement $x := \text{wait } e$, which blocks execution until the task $i \in \text{IDs}$ referenced by e completes, at which point the result which i returns is assigned to the variable x . Furthermore, to facilitate our translations of programs into *sequential* programs with nondeterministically-chosen values, which appear in later sections, we include the **assume** e statement, which proceeds only if the expression e evaluates to **true**, and the nondeterministic assignment $x := \star$.

A *frame* $f = \langle \ell, s \rangle \in \text{Frames}$ is a valuation $\ell : \text{Vars} \rightarrow \text{Vals}$ to procedure-local variables, along with a statement $s \in \text{Stmts}$ describing the entire body of a procedure that remains to be executed; s_p denotes the statement body of procedure p . A *task* $t = \langle i, w, v \rangle$ is an identifier $i \in \text{IDs}$, along with a procedure frame stack $w \in \text{Frames}^*$, and a result value $v \in \text{Vals}$. We say a task $t = \langle i, w, v \rangle \in \text{Tasks}$ is *completed* when $v \neq \perp$, and we maintain that $v = \perp$ if and only if $w = \varepsilon^4$; we refer to t as the *root task* if $i = \perp$. A *task pool* is a set $m \subseteq \text{Tasks}$ in which no two tasks have the same identifier. A *configuration* $c = \langle g, m \rangle \in \text{Configs}$ is a valuation $g : \text{Vars} \rightarrow \text{Vals}$ to the global program variables, along with a task pool m .

To reduce clutter in our definition of program semantics, we define a notion of contexts. A *statement context* S is a term derived from the grammar $S ::= \diamond \mid S; s$, where $s \in \text{Stmts}$. We write $S[s]$ for the statement obtained by substituting a statement

⁴We denote the empty sequence with ε .

SKIP $C[\mathbf{skip}; s] \rightarrow C[s]$	ASSUME $\mathbf{true} \in e(C)$ $C[\mathbf{assume} \ e] \rightarrow C[\mathbf{skip}]$	THEN $\mathbf{true} \in e(C)$ $C[\mathbf{if} \ e \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2] \rightarrow C[s_1]$
ELSE $\mathbf{false} \in e(C)$ $C[\mathbf{if} \ e \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2] \rightarrow C[s_2]$	LOOP $\mathbf{true} \in e(C)$ $C[\mathbf{while} \ e \ \mathbf{do} \ s] \rightarrow C[s; \mathbf{while} \ e \ \mathbf{do} \ s]$	BREAK $\mathbf{false} \in e(C)$ $C[\mathbf{while} \ e \ \mathbf{do} \ s] \rightarrow C[\mathbf{skip}]$
GLOBAL $\frac{x \text{ is a global variable} \quad v \in e(g, M)}{\langle g, M[x := e] \rangle \rightarrow \langle g(x \mapsto v), M[\mathbf{skip}] \rangle}$		
LOCAL $\frac{x \text{ is a local variable} \quad v \in e(g, \ell) \quad \ell_2 = \ell_1(x \mapsto v)}{\langle g, m \cup \{i, \langle \ell_1, S[x := e] \rangle \cdot w, v \rangle\} \rangle \rightarrow \langle g, m \cup \{i, \langle \ell_2, S[\mathbf{skip}] \rangle \cdot w, v \rangle\} \rangle}$		
CALL $\ell \in e(g, T) \quad f = \langle \ell, s_p \rangle$ $\langle g, m \cup \{i, T[\mathbf{call} \ x := p \ e], v \rangle\} \rangle \rightarrow \langle g, m \cup \{i, f \cdot T[x := \perp], v \rangle\} \rangle$		
ASYNC $\ell \in e(g, T) \quad f = \langle \ell, s_p \rangle \quad j \text{ is fresh}$ $\langle g, m \cup \{i, T[\mathbf{async} \ x := p \ e], v \rangle\} \rangle \rightarrow \langle g, m \cup \{i, T[x := j], v \rangle, \langle j, f, \perp \rangle\} \rangle$		
CONTINUE $j = e(g, T) \quad \langle j, -, v \rangle \in m \quad v \neq \perp$ $\langle g, m \cup \{i, T[x := \mathbf{wait} \ e], \perp \rangle\} \rangle \rightarrow \langle g, m \cup \{i, T[x := v], \perp \rangle\} \rangle$		COMPLETE $f = \langle \ell, S[\mathbf{return} \ e] \rangle \quad v \in e(g, \ell)$ $\langle g, m \cup \{i, f, - \rangle\} \rangle \rightarrow \langle g, m \cup \{i, \varepsilon, v \rangle\} \rangle$
RETURN $f = \langle \ell, S[\mathbf{return} \ e] \rangle \quad v \in e(g, \ell)$ $\langle g, m \cup \{i, f \cdot T[x := _], \perp \rangle\} \rangle \rightarrow \langle g, m \cup \{i, T[x := v], \perp \rangle\} \rangle$		

Figure 3.2: Transition rules over program configurations.

s for the unique occurrence of $\diamond$ in S . Intuitively, a context substituted by s , e.g., $S[s]$, indicates that s is the next statement to execute in the statement sequence $S[s]$. Similarly, a *task context* $T = \langle \ell, S \rangle \cdot w$ is a frame sequence in which the first frame's statement is replaced with a statement context, and we write $T[s]$ to denote the frame sequence $\langle \ell, S[s] \rangle \cdot w$. Finally, we write $e(g, \ell)$ (resp., $e(g, T)$) to denote the evaluation of a program expression e given the global and local variable valuations $g, \ell : \mathbf{Vars} \rightarrow \mathbf{Vals}$ (resp., where ℓ is the topmost local variable valuation of T); $e(g, \ell) \subseteq \mathbf{Vals}$ is a *set* of values since program expressions may be nondeterministic, using $\star$.

Figure 3.2 defines an operational program semantics via a set of transition rules on program configurations. The **SKIP** rule just proceeds to the next statement. The **ASSUME** rule proceeds only if the expression e evaluates to true. The **THEN** rule proceed to the then branch if the current valuation of the given expression e evaluates to true, **ELSE** rule proceeds to the else branch if e evaluates to false. The **LOOP** rule iterates the loop if the given expression e evaluates to true, and **BREAK** skips the loop if e evaluates to false. The **GLOBAL/LOCAL** statements set the value of a global/local variable to the value that given expression evaluates to. The **CALL** rule invokes a procedure by adding a new procedure frame on top of the procedure frame stack. The **ASYNC** rule adds a newly-created task to execute an asynchronously called procedure to the task pool, and stores its task identifier (in a procedure-local variable). The **CONTINUE** rule progresses past a **wait** statement when the waited task is already completed, assigning its return value into the result variable. The **COMPLETE** rule completes a task which returns from its bottommost procedure frame, while the **RETURN** assigns the return value of a non-bottom procedure frame to the caller's result variable.

The *initial configuration* $c_0 = \langle g_0, m_0 \rangle$ of a program P is the valuation g_0 mapping each global variable of P to $\perp$, along with a task pool m_0 containing a single root task $\langle \perp, \langle \ell_0, s_{\text{main}} \rangle, \perp \rangle$ such that ℓ_0 maps each variable of the **main** procedure to $\perp$. A *final configuration* $\langle g, m \rangle$ is a valuation g along with a task pool m in which all tasks are completed: for all $\langle -, -, v \rangle \in m$, $v \neq \perp$. An *execution* of a program P to c_j is a configuration sequence $\xi = c_0 c_1 \dots c_j$ starting from the initial configuration c_0 such that $c_i \rightarrow c_{i+1}$ for $0 \leq i < j$; ξ is called *finalized* when c_j is final. We define $R(P)$ as the set of global valuations reached in finalized executions of P , i.e., $R(P) = \{g : c_0 \rightarrow \dots \rightarrow \langle g, - \rangle \text{ is finalized}\}$.

Our definition of the reachable valuations $R(P)$ is purposely restricted to final configurations due to our inclusion of nondeterministic expressions and the **assume** statement, which are needed by our sequentializations in the following sections. This definition of $R(P)$ does not lose any generality since any program can be transformed

into one in which any configuration can reach a completed configuration with the same global valuation, e.g., by adding a `exit` flag to simulate the control flow of an uncaught program exception [Lal et al., 2012].

3.3 The DFW Scheduler

The asynchronous program semantics of the previous section are defined with respect to an implicit task *scheduler*, which enables any non-completed task to execute at any time. Computing the reachable global valuations $R(P)$ of arbitrary programs P is costly. One compelling approach for lowering the cost of program exploration is by considering specialized *delay-bounded* schedulers with limited nondeterminism [Emmi et al., 2011a]. In this section, we provide a formal operational characterization of Emmi et al.’s K -delay bounded depth-first scheduler $DF(K)$ [Emmi et al., 2011a], as well as our novel synchronization-exploiting scheduler $DFW(K)$.

A *scheduler* $\Psi = \langle Q, q_0, \delta, \pi \rangle$ is a set Q of states with initial state $q_0 \in Q$, a transition function $\delta : Q \times ((IDs \times Configs^2) \cup \{\varepsilon\}) \rightarrow Q$, and a task-selection predicate $\pi : Q \rightarrow IDs$. Intuitively, a scheduler state $q \in Q$ determines the task $\pi(q) \in IDs$ that is enabled to make a program transition. A transition $\delta(q, i, c_1, c_2) = q'$ determines the scheduler’s successor state q' to a program transition $c_1 \rightarrow c_2$ of enabled task i from scheduler state q . We represent nondeterminism using ε -transitions $\delta(q, \varepsilon)$; these transitions affect scheduler state only, and not program configuration otherwise. We say Ψ is *deterministic* when $\delta(q, \varepsilon) = q$ for all $q \in Q$.

As an example, we could define a completely nondeterministic scheduler $\langle Q, q_0, \delta, \pi \rangle$, which always enables all pending tasks, with states $Q = IDs^*$ represent scheduling queues, having initial state $q_0 = \perp$; the task-selection predicate $\pi(i \cdot _) = i$ selects the task at the head of the queue. Transitions modify the queue accordingly: enqueueing created tasks j on `ASync` transitions $c_1 \rightarrow c_2$, $\delta(i \cdot I, i, c_1, c_2) = i \cdot I \cdot j$; otherwise not modifying the queue, $\delta(i \cdot I, -, -, -) = i \cdot I$; and rotating the queue on ε -transitions, $\delta(i \cdot j \cdot I, \varepsilon) = j \cdot I \cdot i$. By making a sequence of ε -transitions, this scheduler can enable any previously-created task.

To define the executions admitted by a scheduler Ψ , we make Ψ follow program transitions, and allow Ψ to make ε -transitions at any time. Formally, an Ψ -*execution* is an execution $c_0 c_1 \dots c_j$ such that there exists a sequence $q_0 q_2 \dots q_{j'} \in Q^*$ and a monotonic injection $f : \mathbb{N}^{<j} \rightarrow \mathbb{N}^{<j'}$ such that for each transition $c_i \rightarrow c_{i+1}$ of task u_i , for $0 \leq i < j$, u_i is enabled by Ψ : $u_i = \pi(q_{f(i)})$; and the state of Ψ follows the transition $c_i \rightarrow c_{i+1}$: $\delta(q_{f(i)}, u_i, c_i, c_{i+1}) = q_{f(i)+1}$; additionally, intermediate Ψ -states follow ε -transitions: $q_{i+1} = \delta(q_i, \varepsilon)$ for $0 \leq i < j'$ where $i \notin \text{range}(f)$. Finally we define $R(P, \Psi)$ as the set of global valuations reached in finalized Ψ -executions of P .

We define both the $\text{DF}(K)$ and $\text{DFW}(K)$ schedulers over states which represent the ordered tree of tasks of an execution, in which the children of each node i are the tasks which task i has called asynchronously, in the order in which they are called. Formally, the *Depth-First Scheduler* [Emmi et al., 2011a] is the scheduler $\text{DF}(K) = \langle Q, q_0, \delta, \pi \rangle$ such that

- Q is the set of vertex-labeled trees $\langle V, E, \lambda, d \rangle$ with vertices $V \subset \text{IDs}$, edges E , and labeling function $\lambda : V \rightarrow (\{\text{R}, \text{C}\} \times \mathbb{N})$, assigning each vertex $\lambda(i) = \langle b, k \rangle$ a Ready or Completed status b and a round number $k \in \mathbb{N}$, along with a *delay counter* $d \in \mathbb{N}$.
- q_0 is the tree $\langle \{\perp\}, \emptyset, \{\perp \mapsto \langle \text{R}, 0 \rangle\}, 0 \rangle$.
- $\pi(q)$ is the least, in depth-first order, minimal-round ready vertex as in Figure 3.3(a), and is undefined when q does not contain such a vertex.
- $\delta(q, i, -, -)$ is obtained from q for ASYNC transitions creating task j by adding a rightmost child ($j \mapsto \langle \text{R}, k \rangle$) to the $\langle \text{R}, k \rangle$ -labeled vertex i , as in Figure 3.3(b).
- $\delta(q, i, -, -)$ is obtained from q for COMPLETE transitions by updating i 's label from $\langle \text{R}, k \rangle$ to $\langle \text{C}, k \rangle$.
- $\delta(q, i, -, -) = q$ otherwise.
- $\delta(q, \varepsilon)$ is obtained from q by incrementing the delay counter d , and updating $\pi(q)$'s label from $\langle \text{R}, k \rangle$ to $\langle \text{R}, k + 1 \rangle$, so long as $d < K$; otherwise $\delta(q, \varepsilon) = q$.

Note that at each step of δ , the label of at most one task can change. Furthermore, $\text{DF}(0)$ is deterministic.

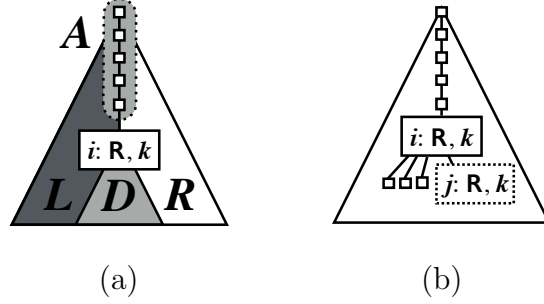


Figure 3.3: (a) A tree of the $DF(K)$ scheduler enabling task i , showing i 's ancestors (A), descendants (D), and the left (L) and right (R) descendants of i 's ancestors. As i is enabled, each node in $A \cup L$ is either completed or has round $> k$, and each node in $D \cup R$ is either completed or has round $\geq k$. (b) When task i posts j , $DF(K)$ adds $(j \mapsto R, k)$ as i 's rightmost child.

Intuitively, $DF(K)$ keeps track of a notion of execution *rounds* from $0 \dots K$ over which tasks execute, and executes lowest-round tasks in depth-first order according to the task tree. For instance, $DF(0)$ allows only a single round of execution, and executes each task in depth-first order until either all tasks are completed, or the task currently enabled by $DF(0)$ blocks. $DF(1)$ executes tasks according to the same order, except that the execution of one single task can increment the delay counter, and be postponed to the second round, resuming if and after which all other tasks complete in the first round. Note that when the currently enabled task in $DF(K)$ is blocked, execution can only progress by advancing the blocked task to a subsequent round, and incrementing the delay counter. As the delay bound $K \in \mathbb{N}$ is increased, the cost of exploration can greatly increase, as $DF(K)$ can allow exponentially more schedules.

To avoid increasing the delay bound $K \in \mathbb{N}$, which exponentially increases the number of alternate schedules explored, and ultimately increases the cost of exploration, we define a scheduler that does not enable tasks that are blocked waiting for others to complete. We define the *Synchronization-Exploiting Depth-First Scheduler* $DFW(K) = \langle Q, q_0, \delta', \pi \rangle$ by extending $DF(K)$ to label each tree node i with an additional Waiting status label, and a counter for the number of tasks that i has posted since its

last-encountered wait statement. (While it is possible to define a “synchronization-exploiting” scheduler without this counter, the resulting sequentialization would be more complicated.) As in $DF(K)$, the task-selection predicate $\pi(q)$ selects the least, in depth-first order, minimal-round ready vertex of q . We define the transition function $\delta'(q, i, c_1, c_2) \stackrel{\text{def}}{=} f(\delta(q, i, c_1, c_2), c_2)$ by the composition of transition function δ with post-processing function $f : (Q \times \text{Configs}) \rightarrow Q$ as follows:

- $\delta(q, i, -, -)$ is obtained from q for **ASync** transitions creating task j by adding a rightmost child ($j \mapsto \langle R, k, 0 \rangle$) to i , as in Figure 3.3(b), and updating i ’s label from $\langle R, k, a \rangle$ to $\langle R, k, a + 1 \rangle$.
- $\delta(q, i, -, -)$ is obtained from q for **WAIT** transitions by updating i ’s label from $\langle R, k, a \rangle$ to $\langle W, k, 0 \rangle$.
- $\delta(q, i, -, -)$ is obtained from q for **COMPLETE** transitions by updating i ’s label from $\langle R, k, a \rangle$ to $\langle C, k, a \rangle$.
- $\delta(q, i, -, -) = q$ otherwise.
- $\delta(q, \varepsilon)$ is obtained from q by incrementing the delay counter d , and updating $\pi(q)$ ’s label from $\langle R, k, a \rangle$ to $\langle W, k + 1, a \rangle$, so long as $d < K$; otherwise $\delta(q, \varepsilon) = q$.
- $f(q, c)$ updates the label of each $\langle W, k, a \rangle$ -labeled vertex i to $\langle R, \max(k, k'), a \rangle$ if and only if (i) i is waiting⁵ for a $\langle C, k', - \rangle$ -labeled task, or i is not waiting and $k = k'$, and (ii) the only $\langle -, \max(k, k'), - \rangle$ -labeled descendants of i are descendants of i ’s a -rightmost children.

In other words, before proceeding past wait statements, the current round of all created subtasks, waited-for or not, are executed. Technically, at each step the label of at most one task can change status from **R** to **W**, though multiple labels can change status from **W** to **R**. Furthermore, $DFW(0)$ is deterministic.

⁵We say i is *waiting for* j in configuration $\langle g, m \rangle$ when $\langle i, T[x := \text{wait } e], - \rangle \in m$ and $e(g, T) = j$.

```

var i: int;

proc p()
  return i

proc main()
  var x: task
  var y: int

  i := 0;
  while * do
    async x := p();
  y := wait x;
  i := i + 1

  return

```

Figure 3.4: A program whose valuations are all reachable in $\text{DFW}(0)$, yet are not all reachable in $\text{DF}(K)$, for any $K \in \mathbb{N}$.

As the following result demonstrates, the $\text{DFW}(K)$ scheduler is strictly more expressive than $\text{DF}(K)$, in the sense that every global variable valuation that can be reached with $\text{DF}(K)$ can also be reached with $\text{DFW}(K)$, for all $K \in \mathbb{N}$, and that for every $K_0 \in \mathbb{N}$, there are programs whose set of valuations reached under $\text{DFW}(K_0)$ cannot be reached by $\text{DF}(K)$ for any finite value $K \in \mathbb{N}$; Figure 3.4 illustrates such a program, whose set of reachable valuations under $\text{DFW}(0)$ is $\{i \mapsto n : n \in \mathbb{N}\}$, while $\text{DF}(K)$ is restricted to $\{i \mapsto n : n \leq K\}$, for any $K \in \mathbb{N}$. While this example may appear artificial at first, web programs that chain asynchronous calls are, in fact, quite common. If the loop in Figure 3.4 were replaced with one that repeats M times, with $M < K$, under the $\text{DF}(K)$ scheduler, it would not be possible to complete program execution at all, since it would not be possible to move past the K -th iteration.

Theorem 1. $R(P, \text{DF}(K)) \subseteq R(P, \text{DFW}(K))$ for all programs P and $K \in \mathbb{N}$; for each $K_0 \in \mathbb{N}$ there are programs P for which $\bigcup_K R(P, \text{DF}(K)) \subsetneq R(P, \text{DFW}(K_0))$.

3.4 Compositional Semantics

Toward simulating the executions under our $\text{DFW}(K)$ scheduler as the executions of a sequential program, we follow Bouajjani et al.'s intuition of *compositional* executions [Bouajjani et al., 2011] with bounded task interfaces. Intuitively, a task interface is a summary of the effect on global storage of one task and all of its subtasks; literally, an interface is a sequence of global valuation pairs, with each pair summarizing a sequence of execution steps of a task and its subtasks. Compositional executions with bounded-size interfaces generalizes various bounding strategies for limiting concurrent behaviors to facilitate efficient program analysis, including context bounding [Qadeer and Rehof, 2005, Lal and Reps, 2009] and delay bounding [Emmi et al., 2011a]. In what follows we specialize Bouajjani et al.'s notion of compositional execution in order to fix a tight correspondence with the executions permitted by our $\text{DFW}(K)$ scheduler.

A $(K+1)$ *round interface* is a map $I : (K+1) \rightarrow (\text{Vars} \rightarrow \text{Vals})^2$ from natural numbers $k \in \mathbb{N} : k \leq K$ to pairs $I(k) = \langle g, g' \rangle$ of global variable valuations; we write $I(k).\text{in}$ to denote g , and $I(k).\text{out}$ to denote g' , and we say I is *fresh* when $I(k).\text{in} = I(k).\text{out}$, for $0 \leq k \leq K$. To compose interfaces, we define a partial composition operator $\oplus$ such that $I \oplus J$ is defined when $|I| = |J|$ and $I(k).\text{out} = J(k).\text{in}$ for all $0 \leq k < |I|$, in which case $|I \oplus J| = |I|$ and $(I \oplus J)(k) = \langle I(k).\text{in}, J(k).\text{out} \rangle$ for all $0 \leq k < |I \oplus J|$. Furthermore, we say an interface I is *complete* when $I(k).\text{out} = I(k+1).\text{in}$ for $0 \leq k < |I| - 1$.

A *compositional configuration* $c = \langle g, w, k, d, I, J \rangle$ is a global valuation $g : \text{Vars} \rightarrow \text{Vals}$, along with a frame sequence $w \in \text{Frames}^*$, a round index $k \in \mathbb{N}$, delay counter $d \in \mathbb{N}$, and interfaces I and J . Figure 3.5 defines a transition relation $\rightarrow$ on compositional configurations, and ultimately an interface generation relation $\leadsto$: the relation $\langle p, v_1, k_1 \rangle \leadsto \langle I, d, v_2, k_2 \rangle$ indicates that procedure p called with argument v_1 in round k_1 , can return the value v_2 . Furthermore, the effect of executing p and all of its subtasks, which executed up until round k_2 having spent d delays, is summarized by the interface I .

$$\begin{array}{c}
\text{CASYNC} \\
\hline
\frac{v_1 \in e(g, T) \quad \langle p, v_1, k_1 \rangle \leadsto \langle J_2, d_2, v_2, k_2 \rangle \quad d_1 + d_2 \leq K}{\langle g, T[\mathbf{async} \ x := p \ e], k_1, d_1, I, J_1 \rangle \rightarrow \langle g, T[x := \langle v_2, k_2 \rangle], k_1, d_1 + d_2, I, J_1 \oplus J_2 \rangle} \\
\\
\text{CWAIT} \\
\hline
\frac{\begin{array}{l} \langle v, k_2 \rangle \in e(g_1, T) \quad g_1 = I[k_1].\mathbf{out} \quad g_2 = J_1[k_2].\mathbf{out} \\ I[k].\mathbf{in} = I[k].\mathbf{out} \text{ for } k_1 < k \leq K \quad J_2 \text{ is a fresh interface} \end{array}}{\langle g_1, T[x := \mathbf{wait} \ e], k_1, d, I, J_1 \rangle \rightarrow \langle g_2, T[x := v], k_2, d, I \oplus J_1, J_2 \rangle} \\
\\
\text{CDELAY} \\
\hline
\frac{d < K \quad g_1 = I[k].\mathbf{out} \quad g_2 = I[k+1].\mathbf{out}}{\langle g_1, w, k, d, I, J \rangle \rightarrow \langle g_2, w, k+1, d+1, I, J \rangle} \\
\\
\text{SUMMARY} \\
\hline
\frac{\begin{array}{l} v_2 \in e(g, \ell) \quad I_1 \text{ and } J_1 \text{ are fresh interfaces} \\ \langle I_1[k_1].\mathbf{out}, \langle v_1, s_p \rangle, k_1, 0, I_1, J_1 \rangle \rightarrow \dots \rightarrow \langle I_2[k_2].\mathbf{out}, \langle \ell, S[\mathbf{return} \ e] \rangle, k_2, d, I_2, J_2 \rangle \end{array}}{\langle p, v_1, k_1 \rangle \leadsto \langle I_2 \oplus J_2, d, v_2, k_2 \rangle}
\end{array}$$

Figure 3.5: The compositional program semantics.

Intuitively, rather than adding a task to the pool, like the **ASYNC** transition of Section 5.3.1, the **CASYNC** rule simply combines the interface J_2 of the asynchronously-called task with the accumulated interfaces J_1 of previously-called tasks. The **CWAIT** rule then, by sequencing the accumulated interface J_1 of previously-called tasks before the current task's interface I , effectively fast-forwards the current task's execution to a point after the execution of the previously-called tasks, and resumes in the round k_2 in which the waited task finished. The **CDELAY** rule simply advances the current task to its next round, spending a single delay. Finally, the **SUMMARY** rule defines the interface generation relation $\leadsto$ as the composition of the task's internal interface I_2 with the accumulated interfaces J_2 of its subtasks.

We then define $\tilde{R}(P, K)$ as the set of global valuations labeling the output of

completed interfaces of the **main** procedure:

$$\tilde{R}(P, K) = \left\{ I[k].\text{out} : \begin{array}{l} \langle \text{main}, \ell_0, 0 \rangle \rightsquigarrow \langle I, -, -, k \rangle, \\ |I| = K+1, \text{ and } I \text{ is complete} \end{array} \right\}$$

This definition allows us to relate the global valuations reachable by executions of $\text{DFW}(K)$ with those reached in our compositional semantics with $(K+1)$ -round interfaces.

Lemma 3.4.1. $R(P, \text{DFW}(K)) = \tilde{R}(P, K)$.

Proof. The lemma states that the asynchronous semantics of a program P under $\text{DFW}(K)$ and its compositional semantics are equivalent. We can show the equivalence between the $\text{DFW}(K)$ execution and the compositional semantics of a program by a structural induction on its task creation tree. The nodes of the task creation tree are the tasks and the parent relation of the tree corresponds to the asynchronous task posting relation, as given in Figure 3.3.

Without the loss of generality, we consider a 1-round execution of P where no tasks are delayed. In a 1-round execution, the vectors of task interfaces are singleton, i.e. only contains the input and output valuations of global variables for a single round. For brevity, we call $I.\text{in}$ and $I.\text{out}$ instead of $I(0).\text{in}$ and $I(0).\text{out}$.

As the basis step, we take a leaf of the tree which is a task J_i without any children tasks. Starting its execution in a global state $g_{J_i}.\text{in}$, J_i runs all its states in a single round and reaches $g_{J_i}.\text{out}$ by applying the asynchronous program transition rules in Figure 3.2. The compositional semantics summarizes the effect of this task as a task interface $J_i = \langle g_{J_i}.\text{in}, g_{J_i}.\text{out} \rangle$.

As the inductive step, we take a parent task which only has m leafs as its children, e.g. the parent task posts m asynchronous tasks none of which post more asynchronous tasks. Applying the base case for all m children tasks, the compositional semantics produces the task interfaces $J_1, J_2, \dots, J_m$ corresponding to their execution in asynchronous semantics. Hence we have,

$J = J_1 \oplus J_2 \oplus \dots \oplus J_m$ that summarizes the effects of all children tasks such that $J.\text{in} = J_1.\text{in}$ and $J.\text{out} = J_m.\text{out}$. Since for each $1 \leq i < m$, $J_i.\text{out} = J_{i+1}.\text{in}$, it

sequences the execution of each task J_i in the order of their creation, e.g. depth first traversal of the task tree.

While summarizing the effect of the children tasks together with their parent tasks, we consider two cases:

- The parent task I does not wait for any of the children tasks.

In this case, the task I runs to completion nonpreemptively. Starting from a initial global state $g_i.\text{in}$, it reaches to $g_i.\text{out}$ by applying the asynchronous program transition rules in Figure 3.2. The compositional semantics summarizes the effect of the execution of I itself as $I = \langle g_i.\text{in}, g_i.\text{out} \rangle$ and the whole execution of I together with its children as $I = I \oplus J = \langle g_i.\text{in}, g_i.\text{out} \rangle \oplus \langle g_{J_1}.\text{in}, g_{J_m}.\text{out} \rangle = \langle g_i.\text{in}, g_{J_m}.\text{out} \rangle$ where $g_i.\text{out} = g_{J_1}.\text{in}$. Since $I \oplus J$ covers the global state values such that $I.\text{out} = J.\text{in}$, the compositional semantics represents the execution where the children tasks are sequenced after the execution of the parent task I , corresponding to the depth first traversal of the task creation tree.

- The parent task I waits for its children task J_i , $1 < i \leq m$ where it has posted m children tasks.

Recall that the *DFW* scheduler schedules the children tasks $J_1, \dots, J_m$ when the parent task I waits for a children task. Consider the task I reaches to a global state $g_i.\text{w}$ when it waits for a child task. The compositional semantics sequence the task interfaces of the children tasks to the parent task interface when the parent task waits for a children. Hence, we have $I = I \oplus J = \langle g_i.\text{in}, g_i.\text{w} \rangle \oplus \langle g_{J_1}.\text{in}, g_{J_m}.\text{out} \rangle = \langle g_i.\text{in}, g_{J_m}.\text{out} \rangle$ where $g_i.\text{w} = g_{J_1}.\text{in}$. Notice that the execution of the child task J_1 starts where I waits and hence suspends in the global state $g_i.\text{w}$. This valuation of the task interface I corresponds to the summary of the execution where I is suspended when it had waited for a children task and afterwards all the children tasks are scheduled in the depth first order of the task creation tree. In the rest of the execution, *DFW* scheduler resumes the execution of I to completion starting from the state $g_{J_m}.\text{out}$. When it

completes, the summarized task interface of I together with its children becomes

$$I = \langle g_i.\text{in}, g_{J_m}.\text{out} \rangle \oplus \langle g_{J_m}.\text{out}, g_i.\text{out} \rangle = \langle g_i.\text{in}, g_i.\text{out} \rangle.$$

The inductive step summarizing a task together with its children task interfaces can be repeated recursively for each node of the task creation tree.

For the sake of simplicity, we showed the equivalence of asynchronous and compositional semantics of a program P with no delaying tasks. The proof applies to multiple rounds of execution as well, where a task interface I as a vector of global state tuples instead of a singleton tuple.

□

Figure 3.6 shows an example for the construction of the interface I that summarizes exactly execution produced by DFW(1) scheduler (given in in Figure 3.6(a)). Consider Task A , that consequently creates tasks B and C , waits for B at the end of segment 1) and waits for C (at the end of segment 4). C is a delaying task (delays after segment 3) and completes after segment 5.

The task A updates the current state it executes in following the standard sequential semantics. Each time it creates a task, it collects and sequences together the interfaces of these created tasks into J (see the sequencing of the interfaces of B and C in Figure 3.6(b)). When it waits for a task, it sequences the accumulated tasks in J into I (as in Figure 3.6(c)). Then, it continues with its internal execution (without preemption to the other tasks). When A waits for the delaying task C , J does not keep new posted tasks to be joined into I , hence I is not updated. But A continues sequencing the rest of its execution (segment 6) in a later round in which C completes. When A completes, it ends up with the interface I that summarizes the execution in Figure 3.6(a).

3.5 Sequentialization

The compositional semantics of the previous section lead to an alternate way to execute asynchronous programs according the DFW(K) using nondeterministic choice (in the

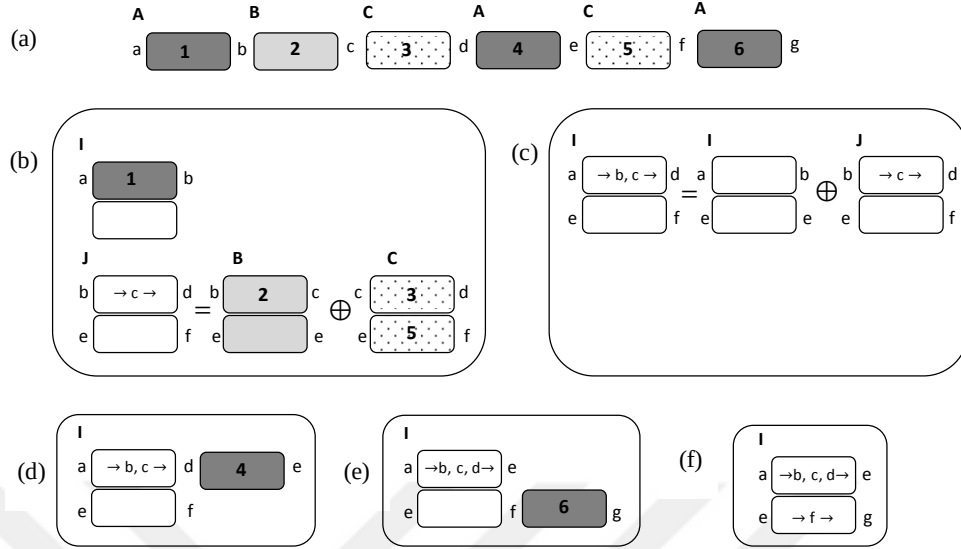


Figure 3.6: Simulating an asynchronous program using compositional semantics.

instantiation of fresh task interfaces): rather than adding asynchronously-called tasks to a task pool, we can simply guess the global states that a task will encounter at the beginning of each of its (up to $K+1$) rounds, and obtain one possible $(K+1)$ -length interface before continuing execution of the calling task. In essence, querying a task for its interface at the point where it is called mimics the same control flow as a procedure call. In this section, we exploit this fact to generate a sequential program $\Sigma(P, K)$ which simulates a given asynchronous program P under the $\text{DFW}(K)$ scheduler; to obtain the interface of an asynchronously-called task, $\Sigma(P, K)$ *calls* the task synchronously, with the nondeterministically-guessed global states constituting the input values of the task's interface. Figure 3.7 lists the statement-by-statement translation $\Sigma(P, K)$ of a program P ; for simplicity, the listed translation assumes that there is one single global variable g ; the extension to multiple global variables is straightforward, by multiplying the **G**, **Guess**, **Next**, and **Save** variables.

Our sequentialization $\Sigma(P, K)$ essentially encodes the interfaces of the previous section using the global **G**, **Guess**, and **Next** variables, along with the **Save** procedure-local variables, and the **Init** constant of the **main** procedure. Initially, the root task,

```

// translation of var g: T
var G[K+1], Guess[K+1], Next[K+1]: T

// new global declarations
var delays: int

// translation of proc p(l: T) s
proc p(l: T, k: K+1)
  var Save: ([K+1]: T) * ([K+1]: T);
  s' // translation of s

// translation of proc main() s
proc main()
  const Init[K+1]: T := G;
  var k: int := 0;
  delays := 0;
  Next := Guess := *;
  s;
  assume G = Guess;
  assume Init[1..K+1] = Next[0..K]

// translation of access to g
G[k]

// translation of call x := p e
call (x,k) := p(e,k)

// translation of return e
return (e,k)

// translation of async t := p e
Save := (G, Guess);
G := Next;
Next := Guess := *;
call t := p(e,k);
assume G = Guess;
G, Guess := Save

// translation of x := wait t
assume G = Guess;
G := Next;
Next := Guess := *;
x, k' := t; k := max(k,k')

// at each possible preemption
if (* && delays < K)
  delays := delays+1; k := k+1

```

Figure 3.7: The K -delay sequentialization $\Sigma(P, K)$.

defined by the `main` procedure, guesses the global values it will encounter at the first point at which it either returns, or waits for a task to complete; this value is stored in both `Next` and `Guess`, and corresponds to the output values of interface I in the compositional semantics of Figure 3.5; the input values of I are stored in `Init`. If the root procedure encounters a `wait` statement, then it validates its `Guess`, advances its state to `Next`, where its previously-called subtasks have left off, and guesses the next global values at which it will either return or encounter a `wait` statement; this process corresponds to composing the I and J_1 interfaces in the `CWAIT` rule of the compositional semantics, effectively sequencing the effects of previously-called tasks before resuming from the `wait` statement.

The other key interesting aspect of $\Sigma(P, K)$ is the translation of the `async` statement. Similar to the sequentialization of the $\text{DF}(K)$ scheduler [Emmi et al., 2011a], the procedure of an asynchronous task is called *synchronously*, using the values `Next` of the global variables effected by previously-called asynchronous procedures; furthermore, the global values guessed to be left behind by the called task are stored into `Next`, from which subsequently-called tasks will resume.

While the global values reachable in the K -delay sequentialization $\Sigma(P, K)$ of a program P are not directly comparable to those of P , since the global variables of $\Sigma(P, K)$ are $(K+1)$ -length vectors of values, we can compare values using a projection function φ mapping $\Sigma(P, K)$'s configurations to values of P . In particular, we define $\varphi(c)$ as $\text{Next}[K](c)$, i.e., the valuation of the `Next` vector's last element in c ; then we define $R_\varphi(P) = \{\varphi(c) : c_0 \rightarrow \dots \rightarrow c \text{ is finalized}\}$. Given this projection, we can show that the set of projected reachable global values in the K -delay sequentialization $\Sigma(P, K)$ of an asynchronous program P is precisely equal to the set of values reachable in P in the K -bounded compositional semantics.

Lemma 3.5.1. $R_\varphi(\Sigma(P, K)) = \tilde{R}(P, K)$.

Proof. (Sketch) For an asynchronous program P , our translation precisely produces the compositional semantics and the set of reachable global values in the K -delay sequentialization $\Sigma(P, K)$ of an asynchronous program P is precisely equal to the set of values reachable in P in the K bounded compositional semantics.

Our sequentialization encodes the interfaces in the compositional semantics using `G`, `Guess`, `Next` and `Save` variables. Essentially, `Next` keeps $J.\text{out}$ and gets updated each time J is joined with a newly posted task. To be able to append the posted tasks after the internal task that creates them, initially `Next` is set to the state it pauses (in either a `wait` or `return` statement) or delays. Since do not know that state initially, we guess the ending states of each round and keep in `Guess`. When we reach to that pause/delay point, we validate the guess.

- The `CASYNC` rule is encoded by the translation of the `async` statement. The translation sets the current state `g` to the `next` state where the last posted task

has finished with (corresponding to the $J.out$). It updates the **next** variable with the new task's ending state which is guessed before its execution and it is verified to be the current state after the execution of the posted task. This precisely captures the generation of a new interface for the new task and joining it with the accumulated task interface J .

- CWait rule is encoded by the translation of **wait** statement. The translation verifies the guess that keeps the pause state of the current task to be the current state and sets the state to **next** that keeps the ending state of the last posted task. This has the same behavior with the CWait rule as it joins the current interface with J (ensuring that the out state of the I is equal to the in state of J). Since the translation does not touch the states $g[i]$ in the intermediate rounds, it satisfies the equality of the **in** and **in** states of these intermediate rounds.
- CDelay rule is encoded in the translation of preemption that increases the current round. Since the program reads the value of the g as $g[i]$ in a given round i , it already operates on the state corresponding to its current round.

□

Combining Lemmas 3.4.1 and 3.5.1, we have our equivalence between the valuations reachable in executions of P under the DFW(K) scheduler with those reachable in executions of the sequential program $\Sigma(P, K)$.

Theorem 2. $R(P, DFW(K)) = R(\Sigma(P, K))$.

3.6 Complexity

While Section 3.3 establishes that DFW(K) generally reaches more program variable valuations than DF(K) does, an obvious concern would be the cost at which it does so. In this section we demonstrate that despite the increased power of DFW(K) with respect to reachability, the essential cost of exploration is roughly equivalent,

in that the reachability problem falls into the same NP-complete class as that of $\text{DF}(K)$. As is standard in the literature, we focus on the effects on complexity arising from concurrency, factoring out effects arising from data; we thus measure the asymptotic complexity of the global-variable value reachability problem assuming program variables have finite domains, and that the number of program variables is fixed. Otherwise, general infinite data domains would lead to undecidability, and the exponential number of valuations of a non-fixed number of program variables would interfere with our complexity measurement. Formally, the $\text{DFW}(K)$ *reachability problem* asks whether a given global program variable valuation g of a given program P is included in $R(P, \text{DFW}(K))$, for a given $K \in \mathbb{N}$, written in unary.

NP-hardness follows directly from the NP-hardness of $\text{DF}(K)$'s reachability problem [Emmi et al., 2011a], since $R(P, \text{DF}(K)) = R(P, \text{DFW}(K))$ for programs P without `wait` statements.

Lemma 3.6.1. *$\text{DFW}(K)$ reachability is NP-hard.*

Our proof of NP-membership reduces the problem to reachability in sequential programs with a fixed number of variables in K . While this amounts to a sort of sequentialization, our sequentialization of Section 3.5 is inadequate, since $\Sigma(P, K)$ has a *linear* number of program variables in K , evaluating to an exponential number of valuations in K . The crux of our proof is thus to design a sequentialization which uses only a *constant* number of additional program variables, independently of K .

Lemma 3.6.2. *$\text{DFW}(K)$ reachability is in NP.*

Our proof of NP-membership reduces the problem to reachability in sequential programs with a fixed number of variables in K . While this amounts to a sort of sequentialization, our sequentialization of Section 3.5 is inadequate, since $\Sigma(P, K)$ has a *linear* number of program variables in K , evaluating to an exponential number of valuations in K . The crux of our proof is thus to design a sequentialization which uses only a *constant* number of additional program variables, independently of K .

Proof. We construct a polynomial time algorithm that verifies a polynomial-sized execution witness in polynomial time. The algorithm reaches its final state iff the provided witness represents a valid $DFW(K)$ execution of a given program P , hence reduces our NP-membership problem to the reachability in this verifier program.

For this aim, we define (i) a witness that represents an $DFW(K)$ execution of a given program P in polynomial size, and (ii) construct a polynomial time algorithm to verify a witness by a translation from tasks' codes. As distinct from our $DFW(k)$ translation that uses $O(K)$ number of variables (keeping K copies of global variables) that yield an exponential number of valuations, we need to build an $O(1)$ translation to conform with the polynomial-time requirement.

We define a witness as a list of task interfaces which are preempted/delayed at least once throughout its execution (which we will refer to a `delaying task`), ordered in the depth-first traversal order of the task creation tree of P . The size of the list (as well as the integer variables used in the translation) is bounded to K due to the delay bound K .

A task interface summarizes the execution of a task together with the execution of the tasks it creates (its child tasks). In other words, the execution in between $g.in[R]$ and $g.out[R]$ states of a round R , encapsulates both the parent and the child tasks' execution in that round. In addition to the $g.in$ and $g.out$ states of each round, task interface keeps the number of child of that task and their indices in their appearance order in the witness, necessary for its verification.

The verifier program (main procedure in Figure 3.8), having multiple assume statements enforce the task interfaces to be sequenced in the $DFW(K)$ order. It completes its execution iff the task interfaces in the witness provide a valid execution when sequenced in that order.

The task interfaces are verified one by one by checking whether the execution of this task yields the $g.in$ and $g.out$ states of its interface and it posts all its child tasks in the given order. A major challenge is to check all rounds of a task and its child tasks in a single call to a task's procedure, without keeping copies of global variables

```

TaskInterface{
  start: K // round the task starts  $\in \{0,1,..K\}$ 
  end: K // round it ends
  val: G // return value
  proc: Proc x E
  gin, gout:  $G^K$ 
  numChildren: K
  children:  $K^{numChildren}$ 
}

// Globals:
var tasks: TaskInterface;
var K: int; // max # of delays
var R: // current round being simulated
var this: TaskInterface; // currently simulated task
var numPosted: K; // current # of children of "this"

// # of children scheduled before "this"
var childrenBeforeMe: K;
var g, next, guess: G

proc main(witness: TaskInterface^K)
  tasks := witness;
  var i, j: int

  i := 0
  while i <= K-1 do
    this := tasks[i];
    R := tasks[i].start;
    g := tasks[i].gin[R];
    numPosted := 0;
    childrenBeforeMe := 0;
    next := guess := *;

    // Call the translation of proc
    call (this.proc)';

    assume guess = g;

    // Verify the interface
    assume numPosted = this.numChildren;
    assume this.end = R;

    // continue the round with the next children
    j := childrenBeforeMe
    while j <= numChildren-1 do
      assume next = tasks[this.children[j]].gin[R];
      next := tasks[this.children[j]].gout[R];
      j := j + 1;

    // End round R
    assume this.gout[R] = next;

    // Simulate next rounds of my children
    if numPosted > 0
      while R < K-1 do
        R++;
        assume this.gin[R] =
          tasks[this.children[0]].gin[R];
        j := 0
        while j <= this.numChildren-1 do
          assume tasks[this.children[j]].gout[R] =
            tasks[this.children[j+1]].gin[R];
          j := j + 1;

        // End the last round
        assume this.gout[R] =
          tasks[this.numChildren-1].gout[R];

        i := i + 1;

      return;

    // Translation of a procedure p(e) s
    proc p(e)
      var save: G * G;
      var i: int
      s'; // translation of s

    // Translation of async t := p e
    if * && (p,e) = this.children[numPosted].proc
      && R = tasks[numPosted].start then
        assume next = tasks[numPosted].gin[R];
        next := tasks[numPosted].gout[R];
        // set the value and the end round of t
        t := (tasks[numPosted].val, tasks[numPosted].end);
        numPosted ++;
      else // Posted task is not a delaying task
        save := (g, guess); g := next;
        guess := next := *;
        call x := p e;
        // Will complete in the current round
        // Set the value and the end round of t
        t := (x, R);
        assume guess = g; <g, guess> := save;

    // Translation of x := wait t:
    (x, t_end) := t;

    if t_end > R
      // If t completes in a later round
      // This round ends in "next"
      assume this.gout[R] = next;

    if numPosted > 0
      // Simulate children rounds up to t_end
      while R < t_end do
        R++;
        assume this.gin[R] = tasks[this.children[0]].gin[R];
        i := 0
        while i <= numPosted - 1 do
          assume tasks[this.children[i]].gout[R] =
            tasks[this.children[i+1]].gin[R];
          i := i + 1;

          assume this.gout[R] =
            tasks[this.children[numPosted-1]].gout[R];

        // Simulate children in round R = t_end and set next
        next := tasks[this.children[0]].gin[R];
        i := 0
        while i <= numPosted - 1 do
          assume next = tasks[this.children[i]].gin[R];
          next := tasks[this.children[i]].gout[R];
          i := i + 1

        // Update g and next
        assume guess = g;
        g := next;
        next := guess := *;

        childrenBeforeMe := numPosted;

    // Translation of preemption:
    assume guess = g;

    // Simulate the children in the current round
    i := childrenBeforeMe
    while i < numPosted - 1 do
      assume next = tasks[this.children[i]].gin[R];
      next := tasks[this.children[i]].gout[R];
      i := i + 1;

    // End the current round
    assume this.gout[R] := next;
    R++;

    // Simulate the children before me
    if childrenBeforeMe > 0 then
      assume this.gin[R] = tasks[this.children[0]].gin[R];
      i := 0
      while i <= childrenBeforeMe - 2 do
        assume tasks[this.children[i]].gout[R] =
          tasks[this.children[i+1]].gin[R];
        i := i + 1

    // Current state is where the children leaves off
    g := tasks[this.children[i]].gout[R];
    next := guess := *;

```

Figure 3.8: Translation to construct the verifier program

for each round. To overcome this, we use the facts that (i) we already have the $g.in$ and $g.out$ states of all rounds of the delaying tasks in the witness and (ii) non-delaying tasks complete in the current round being simulated. We verify a task interface by simulating the execution of that task starting from the $g.in$ of the initial round of a task and during that time we cumulate the effects of the child tasks at task creation points. We use `next` to keep the state in which next task will start with, initially the state where the current task waits, gets delayed (preempts) or returns (at a state where it pauses/completes its execution). The idea is to update `next` using the child interfaces (so that next keeps the effects of tasks to-be-executed at a pause/return point) and updating the current state g using calculated `next` state. Since the parent task does not involve in the rounds later than its completion round, these rounds of the interface are verified by only matching the $g.in$ and $g.out$ states of the child tasks. $\square$

Combining lemmas, we have a tight complexity result.

Theorem 3. *DFW(K) reachability is NP-complete.*

3.7 Empirical Evaluation

We evaluate our DFW(K) scheduler empirically by comparing its sequentialization with an analogously implemented sequentialization of Emmi et al.’s DF(K) scheduler [Emmi et al., 2011a]. Since the DF(K) scheduler does not interpret `wait` statements, we pre-process each program given to the DF(K)-based sequentialization with the translation of Figure 3.9, which outputs an equivalent program without `wait` statements. Essentially, this program keeps track of whether each task has finished using the global `result` variable; the translation of each `wait` statement for a task cannot proceed until its task has completed.

All of our experiments are carried out by applying a sequentialization (either DF(K)’s or DFW(K)’s) on a Boogie [Barnett et al.,] code representation of the input asynchronous program, which is fed to the Corral verification engine [Lal et al., 2012]

```

// new global declarations
var result[int]: T;
var uniqueId: int

// translation of proc p(l: T) s
proc p(l: T, self: int) s

// translation of proc main() s
proc main()
  result := [ $\perp$ ,  $\perp$ , ...];
  uniqueId := 0;
  s

// translation of call x := p e
call x := p (e,0)

// translation of return e
result[self] := e;
return e

// translation of async t := p e
t := ++uniqueId;
async p(e,t)

// translation of x := wait t
x := result[t];
assume x !=  $\perp$ 

```

Figure 3.9: A preprocessing step for the $DF(K)$ sequentialization to remove wait statements.

to detect whether an assertion violation can be reached within a given delay bound K . Our experiments were performed on a typical laptop (Macbook Pro 2013), and we report single-run times. We expect little variation in the comparison between DF and DFW across different hardware configurations, and have observed very little variation in runtime across multiple runs.

Our first set of experiments measures the delay bound and total time necessary to discover assertion violations corresponding to errors reported in a set of C# code fragments found on StackOverflow and MSDN — each around 25-50 LOC. Though we have manually translated the original C# code to Boogie, we have done in a similar to the mechanical translations [Rakamaric and Emmi, 2014].

Lacking an automatic translation from *asynchronous* C# programs, our experiments are tedious to carry out, and are thus limited to a few examples. We note that for programs without wait statements, the verification conditions ultimately generated by both sequentializations are quite similar, and the difference in solving them is insignificant. Experiments from existing works on sequentialization do not consider

programs with wait statements, and are therefore irrelevant to our current study.

Figure 3.10 shows Corral’s execution time to reach each assertion violation in the $DF(K)$ and $DFW(K)$ sequentializations of some real buggy code samples. The benchmarks `CollectionLoad` [Msd,], `SendData` and `Bitmap` [Cod,] are collected from real buggy codes in the online documentation and discussion forums. `SendData` is the benchmark for the buggy code sample summarized in 1.2 in Chapter 1.

In each run, we begin with the delay bound $K = 0$ and increase K until the assertion violation is reachable in the sequentialized program. Our results demonstrate that the $DFW(K)$ scheduler requires consistently fewer delays to reach the assertion violations, which amounts to less exploration time in Corral. The biggest differences appear in the first and third examples, in which the assertion violation is preceded by chains of sequenced asynchronous calls — i.e., where each asynchronous call in the chain is only made after the previous one is waited for; intuitively, each link in this chain forces $DF(K)$ to spend another delay just to progress its execution, whereas $DFW(K)$ ’s natural scheduling order proceeds past each link without spending a delay. These examples illustrate that such call chains are commonplace; even the small bit of code in the third example contains a chain of 5 calls.

In order to validate the efficacy of our delay-bounded sequentialization approach, we have also implemented a “depth-bounded” exploration by translating (by hand) the first `CollectionLoad` example into a sequential program which simulates every asynchronous program execution up to a given number of program steps — we consider that each program statement constitutes one program step. This program’s top-level procedure contains a loop in which each iteration executes a single step of a nondeterministically-chosen task; K iterations of this top-level loop thus simulates all possible asynchronous program executions with up to K steps. Exploration of this program with Corral is intractable: the same bug discovered with $DFW(1)$ requires $K = 9$ program steps, yet Corral is only able to explore up to $K = 4$, in 90 seconds, before timing out at 100 seconds for any depth $K \geq 5$. Note that while $DFW(K)$ is practically limited by the degree K of deviation from $DFW(0)$, of which small values

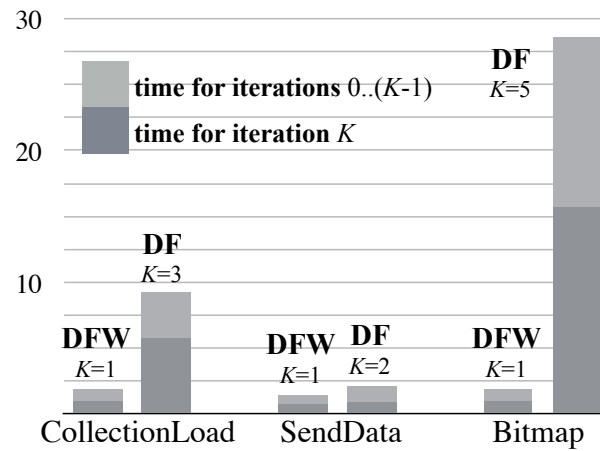


Figure 3.10: Time to bug detection (in seconds for three examples using the DFW and DF sequentializations. Each bar represents the aggregate time over increasing delay bounds, starting from zero, whereas the dark part indicates time spent for the smallest successful delay bound (K).

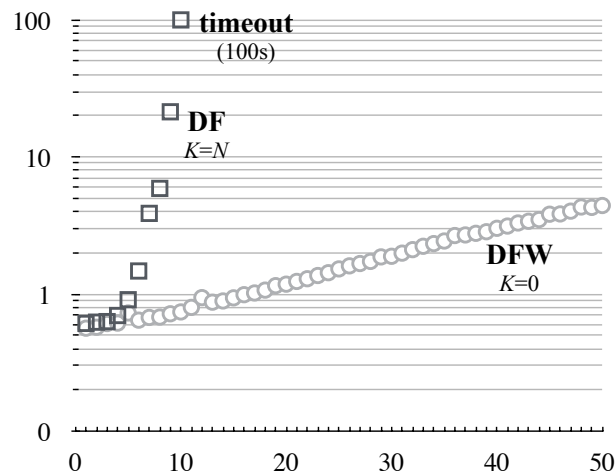


Figure 3.11: Time to bug detection (in seconds) for the parameterized example with N (on the X-axis) chained asynchronous calls. While the DFW sequentialization consistently discovers the bug without delays, DF requires $K = N$ delays, and times out at 100s for $N = 10$.

seem to suffice in exposing concurrency errors, $\text{DFW}(K)$ is not inherently limited by execution depth.

Our second set of experiments attempts to measure the effect of the aforementioned asynchronous call chains on the $\text{DF}(K)$ and $\text{DFW}(K)$ sequentializations using a very simple parameterized program $P(N)$: for each $N \in \mathbb{N}$, $P(N)$ makes N asynchronous calls (to a procedure which simply returns) waiting for each before calling the next, ultimately followed by an assertion violation — i.e., `assert false`. As Figure 3.11 illustrates, the $\text{DFW}(K)$ scheduler never requires a delay to reach the assertion, and its sequentialization scales well, with Corral completing in under 5 seconds even for chains of 50 calls. The $\text{DF}(K)$ scheduler, however, requires N delays for each chain of N calls, and times out at 100 seconds without completing for chains of 10 calls. The utter simplicity of the program $P(N)$ suggests that the $\text{DF}(K)$ sequentialization is limited to very small chains, and ultimately small fragments of synchronization-heavy programs.

3.8 Related Work

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called “sequentialization” [Qadeer and Wu, 2004] explored multi-threaded programs up to one context-switch between threads, and was later expanded to handle a parameterized amount of context-switches between a statically-determined set of threads executing in round-robin order [Qadeer and Rehof, 2005, Lal and Reps, 2009]. La Torre et al. [LaTorre et al., 2010] later extended the approach to handle programs parameterized by an unbounded number of statically-determined threads, and shortly after, Emmi et al. [Emmi et al., 2011a] further extended these results to handle an unbounded amount of dynamically-created tasks, which besides applying to multi-threaded programs, naturally handles asynchronous event-driven programs [Sen and Viswanathan, 2006]. Bouajjani et al. [Bouajjani et al., 2011] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. While others have continued to

propose sequentializations for other bounded concurrent exploration criteria or program models [Kidd et al., 2010, Garg and Madhusudan, 2011, Bouajjani and Emmi, 2012b, Atig et al., 2012, Emmi and Lal, 2012, Emmi et al., 2012a], as far as we are aware, none of these sequentializations are based on a parameterized scheduler which can reduce exploration cost by taking into account program synchronization.

While Emmi et al.’s work [Emmi et al., 2011a] is indeed the starting point for our work, and the syntactic difference between our sequentializations is small, we believe our contribution is significant for the following reasons:

First, and more technically, besides the statements appearing in the translation of the wait statement, our DFW sequentialization must generalize DF. Our translation must repeatedly make guesses — once at each encountered wait statement — for the global state at which begins the sequence of asynchronous tasks called until the next-encountered wait statement (which must be equal to the global state reached by the next-encountered wait statement). In the case of DF, the global state at which the sequence of *all* asynchronous tasks begin is fixed once and for all (and must be equal to the global state reached by main). This extension is subtle, yet crucial.

Second, it is challenging to design a translation which

- (A) correctly preserves causal information flow in the original program, while
- (B) ensuring that the concurrent executions simulated by our sequential program never block because of wait statements.

While the relatively “easy” alternative translation listed in Figure 3.9 does satisfy A, it fails to satisfy B. Our formal development of the DFW sequentialization is a principled way to design a translation which satisfies both Properties A and B: we show that the executions admitted by the DFW scheduler (satisfying B) coincide exactly with our compositional semantics (satisfying A), bridging the gap between any given asynchronous program and its sequentialization.

Finally, comparing to approaches based on dynamic program exploration, while delay-bounding using techniques such as Chess [Musuvathi and Qadeer, 2007] could

capture the same sets of concurrent interleavings for a given delay bound, our static approach promises higher coverage: by using SMT-based symbolic reasoning engines, we can reason about many possible program data values at once, whereas dynamic techniques consider single concrete values. In practice this could allow us to catch data-dependent bugs undetected by a given dynamic technique.



Chapter 4

SYSTEMATIC EXPLORATION OF ASYNCHRONY BUGS IN ANDROID APPS¹

This chapter presents the AsyncDroid tool² for we developed for the systematic discovery and reproduction of asynchrony bugs in Android apps. Given an app and a user-event sequence, AsyncDroid systematically executes alternate schedules of the same asynchronous event handlers, according to a programmable schedule enumerator. The input user-event sequence is given either by user interaction, or can be generated by automated UI “monkeys.” By exposing and controlling the factors which influence the scheduling order of asynchronous handlers, our programmable enumerators can explicate reproducible schedules harboring bugs. By enumerating all schedules within a limited threshold of reordering, we maximize the likelihood of encountering asynchrony bugs, according to prevailing hypotheses in the literature, and discover several bugs in Android apps found in the wild.

4.1 Overview

Android apps execute asynchronously: typically a number of background threads exist to prevent long-running tasks from tying up the main UI thread. Threads execute asynchronously-called procedures concurrently with other threads. Programmers tend to imagine atomically-handled events, without taking all possible thread interleavings into consideration. However, event handlers often call other asynchronous methods, and so the execution of multiple events can interleave. Unexpected schedules of event

¹The work in this chapter is a joint work with Michael Emmi and Serdar Tasiran. [Ozkan et al., 2015]

²<http://github.com/imdea-software/async-droid>

handlers can result in apparently-random bugs which are difficult to reproduce, even given the user-event sequences that trigger them.

In this work we develop AsyncDroid, the first concurrency testing tool for Android apps. AsyncDroid takes a sequence of user events given by user interaction or automated UI “monkeys” and explores different thread interleavings to detect thrown exceptions and assertion violations. Focusing on the systematic exploration of alternate schedules of asynchronously-executing methods, and prioritizing those schedules derived from few re-orderings, our technique uncovers many violations quickly, and uncovers all violations given enough time.

Our prototype implementation explores all deviations from a base schedule within a user-specified bound. In addition to providing a default thread scheduler from which the base schedule can be generated, we also provide an interface allowing users to implement their own scheduler to guide the exploration process along their own insights. Besides control over the thread schedules, our implementation can also record a given sequence of user events, and then replay the same sequence events along alternate thread schedules. We implement both scheduler control and event recording/replaying via program instrumentation, without modifying Android runtime libraries.

4.2 AsyncDroid Design

The basic functionality of AsyncDroid is to repeat a sequence of UI events over a systematic enumeration of thread schedules. We achieve this functionality via a program instrumentation which provides explicit control over thread scheduling, and recording/replaying of UI events for given thread schedules.

4.2.1 AsyncDroid Workflow

AsyncDroid tool is composed of instrumentation and scheduling (Android Schedule Enumerator) libraries together with some support scripts. We inject our Android Schedule Enumerator (Ase) library into the application and link it with the application

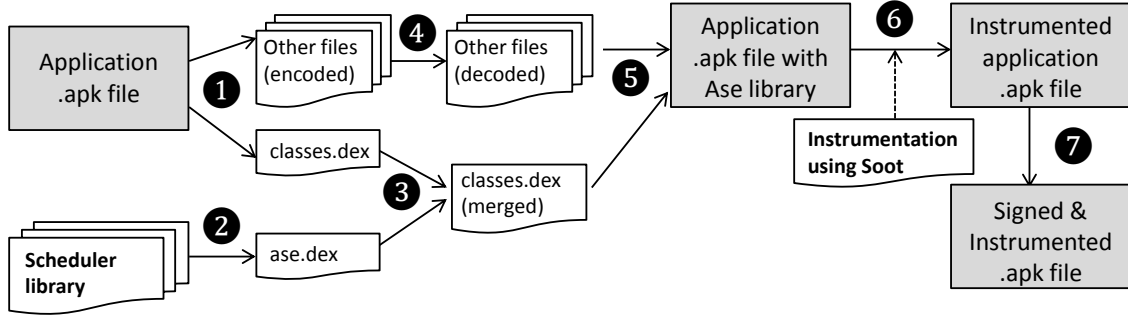


Figure 4.1: The workflow of AsyncDroid

using the instrumentation library. Ase library acts as a scheduler that implements an event recorder, replayer and controls the execution orders of asynchronous tasks.

Figure 4.1 shows the instrumentation workflow of AsyncDroid tool. It takes an application’s apk file, inserts our Ase library in it and instruments the application code with some calls to the Ase library. Then, the instrumented application can be run with record and replay options to record the user events and to replay the recorded events for a set of schedules bounded by the exploration parameter provided.

Our Ase library classes are injected into the application code in `.dex` format. We (1) extract an application `.apk` file to get its `.dex` file, along with the encrypted/compiled versions of some application resource files. (2) We compile our Ase code into `.dex` format using `dx` tool provided in Android SDK. (3) Then, we merge `classes.dex` of the application and `ase.dex` using `dx`. (4) We decrypt the application resource files and (5) build the application `.apk` file from those decrypted files and the merged `.dex` file.

We instrument the built application using Soot framework and insert gluing code into the application (6). When the application `.apk` file is modified, its certificate is lost and the application must be signed again. Finally, we sign the application (7) to make it ready for installation.

The instrumentation modifies the behavior of the application so that it can be launched with record or replay mode depending on the starting intent parameters. A

script file installs the application into a device/emulator and launches it. The script connects the device via Android Debug Bridge (adb) shell. It takes the name of the application under test, launching mode and testing bound as arguments. Then, the instrumented application runs on top of the uninstrumented Android runtime.

4.2.2 Recording and Repeating User Events

To record the user events, we instrument each visible UI component with an additional event handler. When run in record mode, the instrumented handler records each event and forwards it to the original handler. This allows us to capture both direct user interaction, and simulated “monkey” interaction.

In replay mode, we use an input repeater which reads and replays the recorded events for every thread schedule to be tested. The input repeater runs in its own thread and feeds the user events to the application concurrently to the execution of the other threads. AsyncDroid schedules this thread as well as the other application threads, controlling the interleaving between sending an event and the execution in the other threads.

In approaches in the literature, the user events are recorded by saving the coordinates of an input event and replayed by giving the same input on the same coordinates [Gomez et al., 2013]. However, the timing of the inputs differ in each schedule and a UI component might not be visible at a time we want to replay it. Repeating an event using only the coordinates might result in the invocation of a wrong event, since a different view might exist on the original event’s input coordinates at the time of replay. To overcome this, we use an abstraction of an input event close to the application semantics. Our event abstraction keeps the path to UI component of the user event. While replaying an event, we make sure that the recorded path to an event fully matches to the view on the currently visible UI.

4.2.3 Thread Scheduling

AsyncDroid controls the scheduling of the input repeater, UI, and background threads. To explore different execution schedules, AsyncDroid treats the beginnings and ends of asynchronous methods as scheduling points, only preempting threads at these points to determine a complete schedule.

AsyncDroid’s default scheduler runs threads until becoming blocked in a round-robin fashion. The input repeater thread is enabled if it has more events to replay and the next event’s UI component is visible. Similarly, the UI thread and other threads are enabled if they have some tasks to execute. In Android, it is likely that the UI thread have repetitive runtime tasks (for interprocess communication, UI update, etc.) in its queue and never becomes disabled during an execution. In this case, the standard preemption bounding approach would spend a preemption to switch from the UI thread. Our tool blocks a thread and switches to another thread when the message queue of a current thread is empty or it has only recurring Android-runtime messages.

Though the default scheduler runs the threads in round-robin fashion, we use delay bounding [Emmi et al., 2011b] to prioritize our search of alternate executions. For a given bound k , we systematically explore all executions which correspond to thread schedules which are k -delay deviations from the default schedule.

AsyncDroid also allows the programmer to specify his own default scheduler by implementing certain scheduling hooks. To this end, we provide an interface which exposes the current list of application threads, whether each thread is blocked, the list of pending events in the input-repeater thread, and the lists of pending tasks in the UI and background threads. The programmer can access these lists in deciding which thread to dispatch at any given scheduling point.

4.3 Case Studies

As an instructive case study, we investigate an asynchrony bug in the Vville Checker app [Vli,] used with the public bicycle-sharing program in the city of Lille, France.

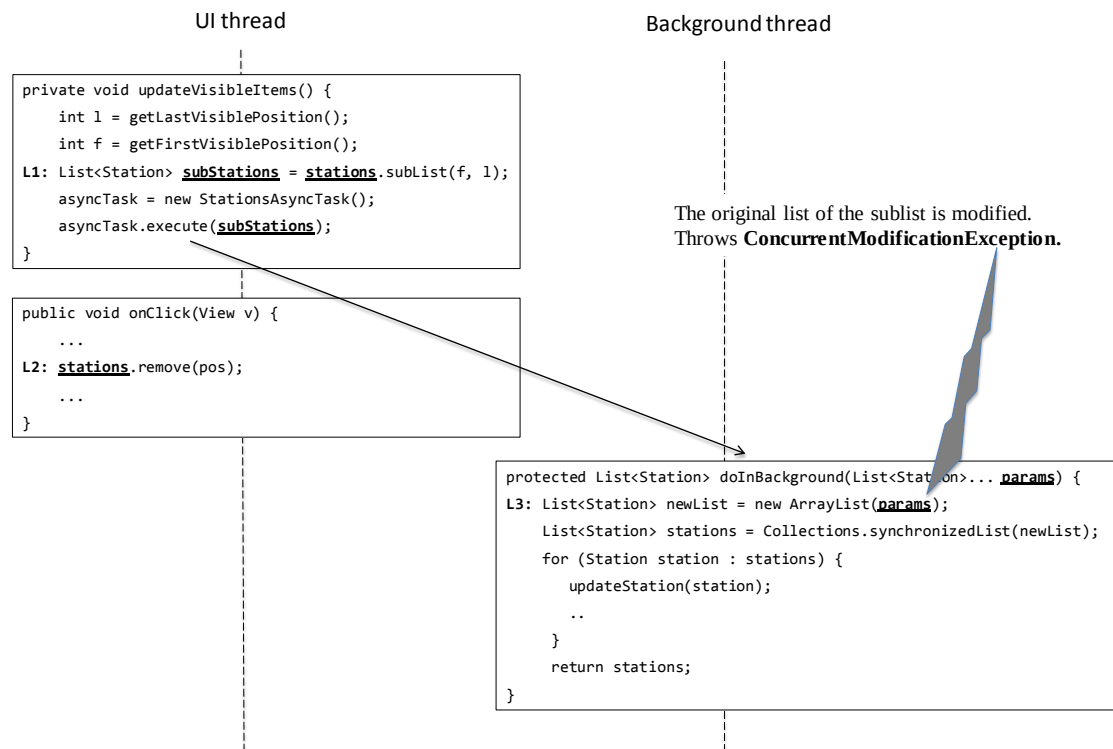


Figure 4.2: An exception thrown only in executions of the Ville Checker app in which the list removal at Statement L2 is executed between the sublist creation at L1 and its use in the constructor at L3 of the asynchronously-called `doInBackground` method.

The app displays a list of bicycle stations together with their status and information. The user can (un)mark a station as a favorite, and limit their view to favorite stations. While the list is being viewed, station information is updated asynchronously in a background thread to keep the UI thread responsive.

The following scenario triggers our bug, depending on the execution order of asynchronous methods. Figure 4.2 shows the relevant application code with distinguished statements labeled L1, L2, and L3.

- The user clicks to view their favorites list.

L1 To initiate the status update of the favorite stations which are currently visible on the screen, the application creates a sublist of visible favorite stations. Crucially

for the bug in question, this sublist is not represented by a new data structure, but is instead backed by the same data structure as the full list of favorite stations.

- The user clicks to remove a station from the favorites list.

L2 An asynchronous task executing on the UI thread removes the station from the favorites list.

L3 An asynchronous task executing on a background thread iterates over the visible favorites list in order to update their statuses.

Since L2 and L3 are executed asynchronously on separate threads, they can execute in any order, depending on hard-to-determine system scheduling factors. In the case that L2 is executed before L3, the `ArrayList` constructor throws a *ConcurrentModificationException* as the favorites list backing the visible favorites sublist has been modified.

To produce the bug, we record the following event sequence and systematically explore possible schedules of asynchronous methods:³

1. click on a menu item to display all stations,
2. click on a station to add it into the favorites,
3. click on a menu item to display the favorite stations, and
4. click on the favorite station to remove it from the favorites.

Note that many schedules of asynchronous methods for this event sequence do not expose the bug. For instance, without incurring delays, `AsyncDroid`'s default scheduler runs each thread until it is no longer enabled before moving on to the next thread in a round-robin fashion. The input repeater thread becomes blocked after Event 3 is actuated, since Event 4 is not enabled until the favorite stations list becomes visible. Next, the scheduler executes the pending UI-thread tasks, causing the asynchronous

³The test which produces this bug is available on `AsyncDroid`'s Github repository.

doInBackground method to become pending on the background thread. Once the UI thread becomes idle, the scheduler executes the background task to completion before returning to the input repeater thread where Event 4 is enabled, the favorite stations list having been updated and made visible. In this way, our default scheduler, without delaying, executes Statement L3 before L2, and does not expose the bug.

However, by enumerating all 1-delay executions, AsyncDroid does discover an execution in which the *ConcurrentModificationException* is thrown, by delaying the background thread before executing Statement L3. This delay causes our default scheduler to return to the input repeater thread where Event 4 is enabled, due to the UI thread having made the favorite stations list visible. After actuating Event 4, we return to the UI thread to process its *onClick* handler before returning to the background thread. This 1 delay execution thus executes Statement L2 before L1, and throws the exception.

We also applied our tool to the ACV comics and image viewer app⁴, the Jamendo online music player app⁵, and a hand-crafted microbenchmark with an injected asynchrony bug. We test the ACV app by providing tap inputs to browse, view, and rotate images. Since tap inputs are always enabled in this app, independently of the app state, our replay is not guaranteed to be faithful to the original event sequence, e.g., in the case that certain taps are ignored in certain states. This limitation could be overcome by more precise tracking of UI state.

The Jamendo music player app is tested by browsing, selecting and playing a radio channel. While playing music, recurring messages are sent to the UI thread to display track progress. As AsyncDroid repetitively runs all schedules without restarting the app, these recurring tasks remain in the message queue after the execution of a schedule completes. This causes the next schedule to start with some leftover tasks. AsyncDroid overcomes this problem by calling an optional finalizer method implemented by the programmer in his app itself to clean up the tasks in the message queues. We tested

⁴<https://github.com/robotmedia/droid-comic-viewer>

⁵<https://github.com/telecapoland/jamendo-android>

	# of events	# of switch	# of sch. dec.	0 delay		1 delay	
				# of conf.	Bug?	# of conf.	Bug?
Vlille Checker	5	9	29	30	No	59	Yes
	7	8	33	34	No	69	Yes
ACV Comic Viewer	6	9	19	14	No	31	No
	8	9	23	19	No	40	No
Jamendo Music Player	3	23	100	69	No	87	No
	5	21	37	24	No	61	No
Microbenchmark	3	5	7	8	No	10	Yes
	5	5	11	12	No	19	Yes

Table 4.1: Quantitative results of our case studies.

the Jamendo app by adding a finalizer method into its source code.

Our hand-crafted microbenchmark inserts into and deletes from a list of items, in response to user events. When the user wants to insert an item, it increases the items count and performs the insertion in a background thread. If the removal of the item in the last index is processed before the background thread, the app throws an *IndexOutOfBoundsException*.

Table 4.1 lists the quantitative results of our case studies. As a rough measure of behavioral coverage, we measure the number of “abstract” program configurations encountered in each exploration, which distinguish only the number of asynchronous tasks pending on each thread. For each run, the table depicts the length of a fixed input event sequence, the number of context switches and the number of scheduling decision points encountered in the zero-delay execution, and the number of abstract configurations encountered. AsyncDroid quickly reproduces the known, yet previously nondeterministically-occurring, bugs in the Vlille Checker and our microbenchmark in a matter of minutes using a single delay. While we do not know whether the ACV and Jamendo apps contain a bug, AsyncDroid does not discover one within a single delay.

4.4 Limitations and Discussion

Applying systematic concurrency exploration to event driven programs faces the fundamental obstacle that some schedules may become infeasible due to the unavailability of a given UI component at a given time. When the component corresponding to a scheduled-for-replay event is not visible on the screen, we delay its activation, disrupting the intended schedule. This limitation raises research questions about how to integrate the treatment of UI events in concert with systematic concurrency exploration.

Controlling all scheduling decisions is a key implementation challenge. In our current prototype, we focus on the systematic analysis of interleavings between the asynchronous methods created by the given app, and leave the scheduling of other asynchronous methods (e.g., periodic system events) uncontrolled.

AsyncDroid currently only supports recording and replaying of certain types of UI events: we handle simple clicks, but not text inputs nor gestures. Capturing a wider set of UI events will allow us to test a larger set of applications. Another direction of future work also involves developing coverage metrics to evaluate how various scheduling strategies compare with respect to coverage of program behaviors.

4.5 Related Work

Existing approaches to bug detection in Android apps fall into two basic categories. The first category focuses on UI input testing [Anand et al., 2012a, Azim and Neamtiu, 2013a, Choi et al., 2013, Machiry et al., 2013a]. Orthogonally to these techniques, which test a single execution of any given UI-event sequence, our goal is to explore the alternate schedules of execution for a given UI-event sequence, thus uncovering elusive concurrency-related bugs. The second category of techniques investigates race detection [Maiya et al., 2014, Hsiao et al., 2014, Bielik et al., 2015, Lin et al., 2014]. Our work is complementary to these techniques and it is novel in two respects: it is a dynamic analysis rather than static, and it does not report false positives.

While prioritized systematic exploration of concurrent program executions has been studied before [Musuvathi and Qadeer, 2007, Emmi et al., 2011b], the adaptation to event driven programs poses some specific difficulties. A tool must explore different possible concurrent behaviors for a given, fixed user interaction with the program. This requires the recording and replay of user input events while exploring alternate schedules. Moreover, it is nontrivial to design effective thread schedulers for the typical Android “looper” threads, which do nothing but execute the handlers of received messages.



Chapter 5

CHARACTERIZATION OF THE CORRECTNESS OF EVENT-DRIVEN ASYNCHRONOUS PROGRAMS: ROBUSTNESS AGAINST ASYNCHRONY¹

In this chapter, we characterize the correctness for event-driven asynchronous programs as *robustness against asynchrony* that (i) meets the programmer expectations, (ii) captures the bugs in the wild as robustness violations and (iii) is algorithmically checkable.

5.1 Introduction

A common rule of thumb for achieving responsiveness event-driven asynchronous programs is to perform tasks that are guaranteed to complete in a short time (e.g. updating UI components) in the main thread and delegating long running tasks by asynchronously invoking them on some background threads. While this style simply guarantees efficiency and responsiveness to user events, it is actually deceptively simple. One tends to think that the events are handled in isolation, without interference from other event handlers or similarly the background tasks run in isolation without interference from other concurrent tasks. It is also easy to make certain assumptions about the relative completion order of tasks or arrival of responses.

Our notion of robustness bridges the gap between the programmer expectations and the possible timings in the execution of asynchronous programs. Asynchronously robust programs meet the programmer expectations for the execution of the program in the existence of the parallel execution of the concurrent tasks and the interference

¹The work in this chapter is a joint work with Ahmed Bouajjani, Constantin Enea, Michael Emmi and Serdar Tasiran.

on the shared memory.

In our programming abstraction, there are two views for an event-driven shared memory asynchronous program: a high-level and a low-level view.

- The high-level view is a reference model whose purpose is to reason about functional correctness. In this view, an event-driven asynchronous program consists of a single thread handling user events in a serial manner, one after the other. Each event is handled by executing its corresponding code until completion, while the asynchronous procedure invocations are postponed and executed in a recursive manner afterwards, one after the other, in the order of their invocation.
- The low-level view reflects the concurrency of the actual program running on the mobile platform. It is more highly and explicitly concurrent. The main thread and asynchronous tasks posted to background threads interact over shared memory.

The high-level view is simpler to apprehend, while the lower-level view corresponds to adopting optimizations in the sake of improving performance and responsiveness, removing more of the work from the UI thread and delegating it to the asynchronous tasks running on the background threads.

For reasoning about functional correctness on the high-level model to carry over to the implementation, the low-level model must be a *refinement* of the high-level model in a way we formally define. Roughly, a program should have the same set of observable reachable states whether it was running under the (weak) semantics corresponding to the low-level view or under the (strong) semantics corresponding to the high-level view. To ensure this refinement, the low-level view must avoid certain kinds of interference (e.g., two threads writing to the same location). A program that satisfies this refinement condition is said to be *robust against asynchrony* in the sense that the concurrency introduced in the low-level view does not change the observable results of its behaviors.

Besides capturing common informal program expectations, robustness against asynchrony enables an effective decomposition of the difficult problem of asynchronous program verification. Robustness against asynchrony can be proved to be equivalent to the conjunction of two correctness criteria that we call *event-serializability* and *event determinism* of the considered program.

Event serializability captures the property that each interleaved execution of events coincides with some serial execution of events, and Event determinism captures the property that all interleaved executions of the asynchronous-procedure invocations of a single event lead to the same unique result.

To define *event serializability*, we consider for each event handler its action set, i.e., all the actions performed by asynchronous procedures it calls and threads it creates. Event serializability then requires that executions of event-driven asynchronous programs be equivalent to those obtained by running the entirety of an event's action set, followed by the next event's action set, and so on. This requirement intuitively corresponds to the expectation that event handlers run without interference from each other. Assuming event serializability allows the programmer to ignore other event handlers when analyzing the code of a particular event handler. Event serializability can be ensured, for instance, by ensuring that each execution of an event handler accesses data disjoint from other handlers, or by ensuring that only one handler of each type of event may be active at any moment. Differently from some other settings, in an event-driven asynchronous program, an event-handling procedure typically creates asynchronous tasks to be run on possibly multiple background threads and these tasks may continue execution even after the event-handling procedure itself has returned. Some event handlers may create regular threads to carry out specialized tasks. While various notions of serializability have been studied in other contexts, the view of a mobile application as an event driven program with serializable event-handlers, and empirical support for the appropriateness of this view is novel in this work.

The *event-determinism* correctness criterion focuses on the actions performed by a single event handler and potential interference among the asynchronous tasks

and threads the event handler creates. Event determinism requires for each event handler invocation, starting from a given state, regardless of how actions in its action set get interleaved, that the invocation leads to a deterministic state update. For event-serializable programs, event determinism may be ensured by ensuring that each asynchronous task appears to run without interference, for instance, by scheduling all asynchronous tasks on the same OS thread in a particular order. Assuming event determinism frees the programmer from considering interference among asynchronous tasks invoked by the same event handler. Non-interference of asynchronous procedure invocations frees the programmer from considering interference among asynchronous tasks invoked by the same event handler.

In the next section, we provide motivating examples and empirical evidence from the literature and software in the wild to support and justify these criteria. We find that violations of event serializability and determinism closely and accurately correlate with concurrency bugs in event-driven shared memory asynchronous programs.

5.2 Motivating Robustness as Correctness Criteria

In this section, using concurrency bugs reported in open-source repositories, we illustrate the correspondence between real bugs and event-serializability and determinism violations. Also, we provide a typical example of a program satisfying both criteria, and therefore, our notion of robustness.

5.2.1 A Violation to Event Serializability

We illustrate a violation to event serializability using a real code excerpt from the IrcCloud app [Irc, a] for chatting on the IRC listed in Figure 5.1. The user event of pressing the “send” key is handled by the procedure `onKey`. Actions associated with this event handler include actions performed by `onKey` on the main (UI) thread, the creation of the asynchronous `SendTask` task, actions performed by `SendTask.onPreExecute()` on the UI thread before the actions performed on a background thread by `SendTask`’s long-running `doInBackground` procedure. Another event handler in this example is

```

class SendTask extends AsyncTask {
    void onPreExecute(){
        e.command = msgTxt.getText();
    }

    void doInBackground(..) {
        if(...) {
            /* corrected version:
            write e.command into JSON object */
            write msgTxt.getText()
            into JSON object
        }
    }
}

ActionEditText msgTxt;
boolean onKey(...) {
    // actions on UI thread
    new SendTask().execute();
}

void onDoubleClicked(String name) {
    text += " " + name;
    msgTxt.setText(text);
}

```

Figure 5.1: A program with event-serializability violation.

`onDoubleClicked`, which appends to the message text the name `name` of the user whose name is clicked on.

Event serializability requires that all executions of a program be “equivalent” to one in which, for each user event, all the program actions (transitively) performed by its handler are executed without interference by handlers of other events.

To see an event-serializability violation, let us consider a scenario where the user types “Hello”, presses “send”, and then double-clicks on another IRC user’s name. Suppose that `onDoubleClicked` starts running on the UI thread while `SendTask.doInBackground` is in progress on the background thread. These two procedures’ accesses can interfere with each other. In particular, the ordering of `msgTxt.getText()` with respect to the appending of `name` to `msgTxt` determines whether “Hello” or “Hello foo” gets sent on the network. Moreover, since `onKey` first records `msg.getText()` to a field `e.command`, an execution of these two events can end in a program state in which `e.command` contains “Hello” while `msgTxt` contains “Hello foo”. This end state is not consistent with *any* serial execution of these two event handlers. This behavior was reported as a bug, and the code was updated [Irc, b] so that `e.command` (instead of `msgTxt`, which

may have changed) is written into a JSON object and sent on the network. It was the designers' intent for the entire event handling code for the "send" key to appear atomic.

5.2.2 Violations to Event Determinism

5.2.2.1 Event nondeterminism with a single background thread

Figure 5.2 lists an example [App,] where an asynchronous procedure executing on a background thread (the first *runnable*) interferes with another asynchronous procedure executing on the UI thread (the second *runnable*).

The procedure `iconPackUpdated` creates an asynchronous task (the first runnable to be executed by the created thread) to initialize the `mAppRowAdapter` object. Then it creates another task, to be run by the UI thread, that uses `mAppRowAdapter` to update the list view of displayed icons. In an effort to ensure that the second task runs after the first task completes, the programmer posts the second task to the UI thread after a second's delay.

Due to scheduling decisions, other concurrent tasks, or the first runnable code taking long, it is possible for the first task not to complete even after a second. In this case, the second runnable code will experience a null pointer exception, while in other schedules, the code works as intended. Although the programmer had intended a deterministic outcome of this event handler there are executions with different outcomes, including errors.

5.2.2.2 Event non-determinism leading to security issues

Figure 5.3 presents an example of an event handler that violates event determinism. This example is a simplified version of a visual security vulnerability identified in an earlier version of Internet Explorer [Chen et al., 2007]. The original example concerns asynchrony in a web browser, but similar scenarios are conceivable in other, e.g., mobile applications. The user event in this example is the loading of a web page (e.g., `paypal.com` as in [Chen et al., 2007]). In the infected version of the web page, the

```

void iconPackUpdated() {
    new Thread( new Runnable() {
        void run() {
            ...
            mAppRowAdapter = new AppRowAdapter(...);
        }
    } ).start();

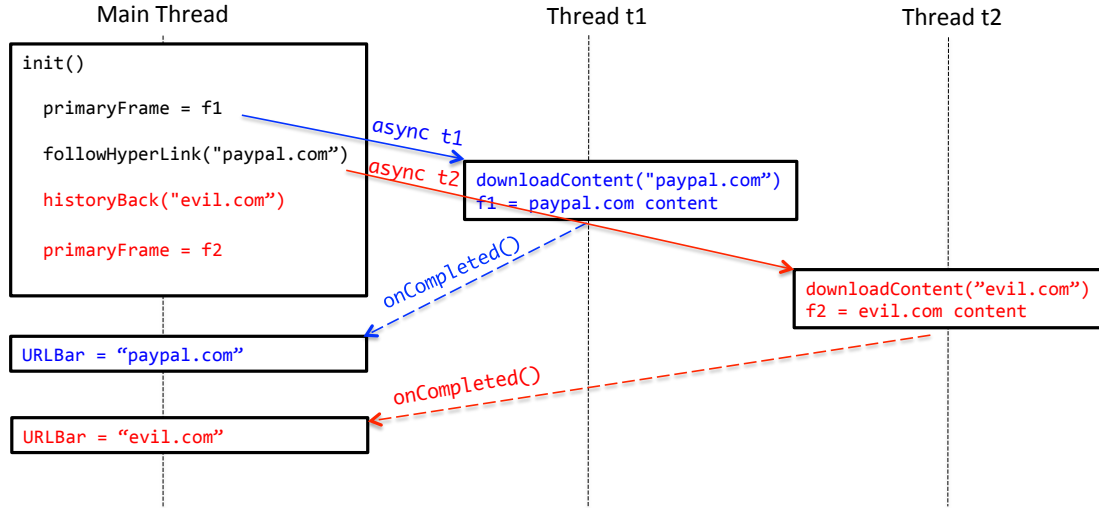
    new Handler().postDelayed( new Runnable() {
        void run() {
            ...
            if(setAdapter)
                listView.setAdapter(mAppRowAdapter);
            ...
            mAppRowAdapter.notifyDataSetChanged();
        }
    }, 1000);
}

```

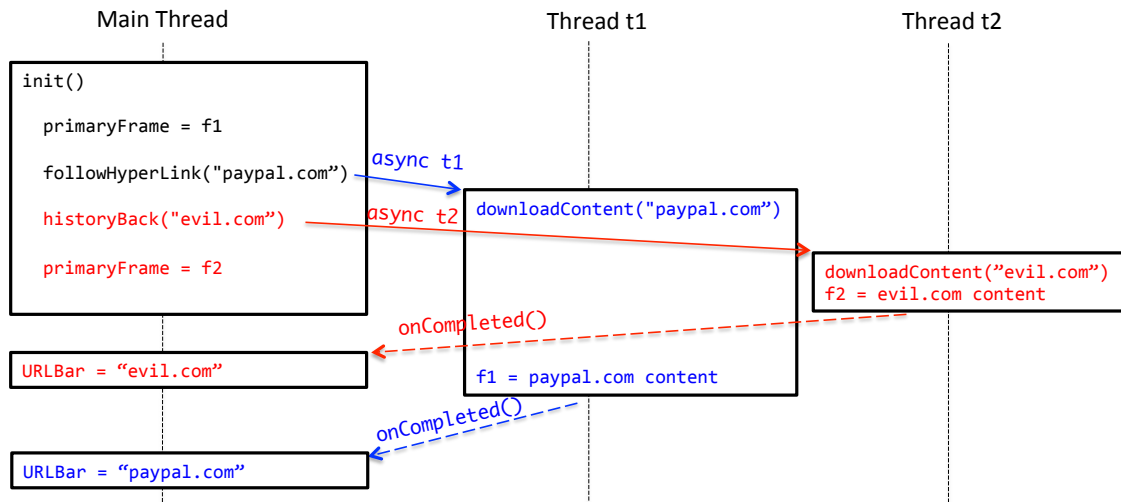
Figure 5.2: A program with event-determinism violation.

loading event has non-deterministically interfering asynchronous procedures intended to cause a security problem that only occasionally occurs, thus, avoids easy detection. When accesses by concurrent asynchronous procedures are interleaved in a particular way, the attack succeeds (the scenario in Figure 5.3b). In a succeeding attack, the content of the web page displayed belongs to `evil.com` whereas the URL bar of the browser shows `paypal.com`, leading the user to think that he is interacting with `paypal.com`. In other interleavings, the attack fails (the scenario on the left) and both the content and the URL bar display information from `evil.com`. The non-determinism arises from different possible execution orders of asynchronous tasks created by the single event handler `processUserRequest`. Detecting the event-determinism violation using our approach uncovers this attack/bug.

The actual scenario in [Chen et al., 2007] includes cases in which the certificates and other browser security features also appear associated with `paypal.com` although the displayed content is `evil.com`. EXECUTION 1 and EXECUTION 2 are two executions



(a) After this execution, the final state has URLBar == "evil.com", primaryFrame == f2 and f2 == evil.com content



(b) This is an execution with security attack. The final state has URLBar == "paypal.com", primaryFrame == f2 and f2 == evil.com content

Figure 5.3: Two executions of an event leading to different final states. The browser uses two "frames" f1 and f2 in which web pages can be rendered. Setting primaryFrame to f1 or f2 selects which frame is displayed. The other frame becomes invisible.

```

List news;

// Event 1
void searchForNews(String key) {
    new SearchTask.execute(key);
    new SaveKeyTask.execute(key);
}

class SearchTask extends AsyncTask {
    List result = null;

    void doInBackground(String key) {
        // connect to the network
        result = ...
    }

    void onPostExecute() {
        list = result;
        // display the list of titles
    }
}

class SaveKeyTask extends AsyncTask {
    void doInBackground(String key) {
        // write key to the database
    }
}

// Event 2
// shows the detail of the idth news
void showDetail(int id) {
    new DownldTask.execute(id);
}

class DownldTask extends AsyncTask {
    String content = null;

    void doInBackground(int id) {
        // connect to the network
        // and set content
        content = ...
    }

    void onPostExecute() {
        // display the content
    }
}

```

Figure 5.4: A robust program sample.

of the single event handler `processUserRequest` that lead to two different final states, i.e., this program has an event-determinism violation leading to a security issue. This lends intuitive support for our claim that concurrency-caused nondeterminism in the execution of a single event handler is unintended and undesirable.

5.2.3 A Robust Program

The program in Figure 5.4 has two event handlers `searchForNews` and `showDetail` which can be invoked by the user to search for news containing a keyword and to dis-

play the details of a selected news respectively. The `searchForNews` procedure creates two asynchronous tasks using the `AsyncTask` class defined in the Android library. An `AsyncTask` object contains `onPreExecute`, `doInBackground` and `onPostExecute` methods that can be implemented by the programmer. Once an `AsyncTask` is executed, its `onPreExecute` method runs synchronously on the main (UI) thread and when it completes, `doInBackground` (that typically contains long running code) is asynchronously executed on a background thread. After the procedure in `doInBackground` completes, `onPostExecute` (that typically updates UI) method is run asynchronously on the main thread.

When the user input to search for news is triggered, `SearchTask` and `SaveTask` are executed. The `searchTask` connects to the network, searches for the keyword and fetches the list of resulting news titles asynchronously in a background thread. Then, it displays the list of titles to the user. `SaveTask` saves the keyword to a database representing the search history in the background.

The execution of the background tasks `SearchTask.doInBackground` and `SaveTask.doInBackground` might be concurrent or serial to each other depending on the Android version or explicit programmer annotations. The event `searchForNews` is deterministic, i.e., any interleaving of the created tasks produces the same result.

The second event, to show the details of a selected news title, can be triggered once the list of titles are displayed on the screen. It invokes an asynchronous task to download the contents of the news in the background and then displays it. In this case, the tasks are executed in a fixed order and the event is trivially deterministic.

Concerning serializability, note that the execution of `SaveTask` in the first event and the execution of the second event might interleave. We assume that the second event is triggered once the results are displayed to the user. However, any such interleaving results in the same state as a serial execution of the two events.

5.3 Event-Driven Asynchronous Programs

In order to give generic characterizations of robustness, serializability, and determinism which do not depend on any particular asynchronous-programming platform or syntax, we frame our discussion around the abstract notion of programs defined in Section 5.3.1, viewing programs as transition relations over abstract global and local states. For illustrative purposes, and Section 6.1's reduction via program translation, we also give a canonical program syntax. Sections 5.3.3 and 5.3.2 then give two alternative weak and strong semantics to programs. We restrict our attention to programs that are *data-deterministic*, in the sense that the evaluation of every (boolean) expression is uniquely determined by the variable valuation.

5.3.1 Program Syntax and Semantics

We fix sets G and L of global and local program states, a set $X = \{\text{rd}(x), \text{wr}(x) : x \in \dots\}$ of memory accesses, a set $W = \{\text{main}, \text{any}\}$ of invocation places, and a set

$$B = \{\text{invoke}(\ell, w) : \ell \in L, w \in W\} \cup \{\text{return}\} \cup X \cup \{\varepsilon\}$$

of program actions, where ε represents irrelevant actions. Local states $\ell \in L$ represent the code and data of an asynchronous procedure invocation, including the code and data of all nested synchronous procedure calls. The $\text{rd}(x)$ and $\text{wr}(x)$ represent read and write accesses to variable x ; $\text{invoke}(\ell, w)$ represents an asynchronous invocation whose initial local state is ℓ ; the invocation is to be run on a distinguished main thread when $w = \text{main}$, and on an arbitrary thread when $w = \text{any}$. Finally, the return action represents the return from an asynchronous procedure invocation.

A program $P : G \times L \rightarrow G \times L \times B$ maps global states $g \in G$ and local states $\ell \in L$ to new states and program actions; each $P(g, \ell)$ represents a single program transition. We assume that when b is a call or return action and $P(g, \ell) = \langle g', -, b \rangle$ then $g = g'$. An *event set* $E \subset L$ is a set of local states; each $e \in E$ represents the code and data for a single event.

$x := y \quad y := x \quad \text{assume } y \quad \text{call } p(\mathbf{y}) \quad \text{async}[w] \, p(\mathbf{y}) \quad \text{return}$

Figure 5.5: A canonical program syntax. The metavariables x and y range over global and local variable names, respectively, p ranges over procedure names, and w over the symbols “main” and “any.”

Canonical Program Syntax. Supposing that the global states $g \in G$ are maps from program variables x to values $g(x)$, and that local states $\ell \in L$ map program variables y to values $\ell(y)$ and a program counter variable pc to program statements $\ell(\text{pc})$, we give an interpretation to the canonical program syntax listed in Figure 5.5. For simplicity, we omit the interpretation of synchronous procedure calls $\text{call } p(\mathbf{y})$ which is defined as usual. For instance, writing ℓ^+ to denote $\ell[\text{pc} \mapsto \ell(\text{pc})+1]$, then $P(g, \ell)$ is

- $\langle g[x \mapsto \ell(y)], \ell^+, \text{wr}(x) \rangle$ when $\ell(\text{pc})$ is a global-variable write $x := y$,
- $\langle g, \ell^+[y \mapsto g(x)], \text{rd}(x) \rangle$ when $\ell(\text{pc})$ is a global-variable read $y := x$,
- $\langle g, \ell^+, \text{rd}(y) \rangle$ when $\ell(\text{pc})$ is **assume**(y) and $\ell(y) \neq 0$,
- $\langle g, \ell, \varepsilon \rangle$ when $\ell(\text{pc})$ is **assume**(y) and $\ell(y) = 0$,
- $\langle g, \ell^+, \text{invoke}(\ell', w) \rangle$ when $\ell(\text{pc})$ is an asynchronous invocation $\text{async}[w] \, p(\mathbf{y})$, where ℓ' maps the parameters of procedure p to the invocation arguments $\mathbf{y}$ and pc to the initial statement of p , and
- $\langle g, \ell, \text{return} \rangle$ when $\ell(\text{pc})$ is the **return** statement.

The semantics of other statements, including if-then-else conditionals, while loops, goto statements, synchronous procedure calls, etc., is standard, and yield the empty action ε .

An event is called *sequential* when its code doesn't contain asynchronous invocations `async[w] p(y)`. Also, a program P with event set E is called *sequential* when every event $e \in E$ is sequential. Otherwise, P is called *concurrent*.

5.3.2 Weak Asynchronous Semantics

Our weak semantics maximizes the set of possible program behaviors by allowing events to interleave and interfere with each other, and allowing asynchronous procedure invocations to execute on separate threads, and not necessarily in invocation order. Configurations thus maintain sets of running procedure invocations as well as an unordered queue of pending invocations, and invocations are associated with events and threads.

In order to characterize executions by the event-serializability and event-determinism criteria of Section 5.4.2, we expose the following set A of actions in execution traces:

$$A = \{\text{start}(j), \text{end}(j) : j \in \mathbb{N}\} \cup X \\ \cup \{\text{invoke}(i), \text{begin}(i), \text{return}(i) : i \in \mathbb{N}\}$$

By convention, we denote asynchronous procedure invocation, event, and thread identifiers, respectively, with the symbols i, j, k . The $\text{start}(j)$ and $\text{end}(j)$ actions represent the start and end of event j ; the $\text{invoke}(i)$, $\text{begin}(i)$, and $\text{return}(i)$ actions represent an asynchronous procedure invocation (when it is added to the queue of pending invocations), the start of i 's execution (when it is removed from the queue), and return of i , respectively. The set X of memory accesses is defined as in the program actions of Section 5.3.1.

A *task* $u = \langle \ell, i, j, k \rangle$ is a local state $\ell \in L$ along with invocation, event and thread identifiers $i, j, k \in \mathbb{N}$, and U denotes the set of tasks. We write $\text{invoc}(u)$, $\text{event}(u)$, and $\text{thread}(u)$ to refer to i , j , and k , respectively. A *configuration* $c = \langle g, t, q \rangle$ is a global state $g \in G$ along with sets $t, q \subseteq U$ of running and waiting tasks such that: (1) invocation identifiers are unique, i.e., $\text{invoc}(u_1) \neq \text{invoc}(u_2)$ for all $u_1 \neq u_2 \in t \cup q$, and (2) threads run one task at a time, i.e., $\text{thread}(u_1) \neq \text{thread}(u_2)$ for all $u_1 \neq u_2 \in t$. The set of configurations is denoted by C_w . We say that a

$$\begin{array}{c}
\text{EVENT} \\
\frac{e \in E \quad i, j \text{ are fresh}}{g, t, q \xrightarrow{0, \langle -, j, \text{start}(j) \rangle} g, t, q[\langle e, i, j, 0 \rangle]} \quad \frac{\text{DISPATCH} \quad u = \langle \ell, i, j, k \rangle \quad k \text{ is idle}}{g, t, q[u] \xrightarrow{k, \langle i, j, \text{begin}(i) \rangle} g, t[u], q} \\
\\
\text{CALL} \\
\frac{P(g, \ell_1) = \langle -, \ell'_1, \text{invoke}(\ell_2, w) \rangle \quad u_2 = \langle \ell_2, i_2, j, k_2 \rangle \quad i_2 \text{ is fresh} \quad k_2 \text{ is 0 if } w = \text{main or fresh otherwise}}{g, t[\langle \ell_1, i, j, k \rangle], q \xrightarrow{k, \langle i, j, \text{invoke}(i_2) \rangle} g, t[\langle \ell'_1, i, j, k \rangle], q[u_2]} \\
\\
\text{RETURN} \\
\frac{P(g, \ell) = \langle -, -, \text{return} \rangle \quad j \in \{\text{event}(u) : u \in t \cup q\}}{g, t[\langle \ell, i, j, k \rangle], q \xrightarrow{k, \langle i, j, \text{return}(i) \rangle} g, t, q} \\
\\
\text{END EVENT} \\
\frac{P(g, \ell) = \langle -, -, \text{return} \rangle \quad j \notin \{\text{event}(u) : u \in t \cup q\}}{g, t[\langle \ell, i, j, k \rangle], q \xrightarrow{k, \langle i, j, \text{return}(i) \rangle} g, t, q \quad k, \langle i, j, \text{end}(j) \rangle} \\
\\
\text{LOCAL} \\
\frac{P(g, \ell) = \langle g', \ell', a \rangle \quad a \in \{\varepsilon, \text{read}(x), \text{write}(x)\}}{g, t[\langle \ell, i, j, k \rangle], q \xrightarrow{k, \langle i, j, a \rangle} g', t[\langle \ell', i, j, k \rangle], q}
\end{array}$$

Figure 5.6: The weak transition function $\rightarrow$ determined by a program P and event set E .

thread k is *idle* in c when $k \notin \{\text{thread}(u) : u \in t\}$, and that an identifier i, j, k is *fresh* when $i, j, k \notin \{\alpha(u) : u \in (t \cup q)\}$ for $\alpha \in \{\text{invoc}, \text{event}, \text{thread}\}$, respectively. A configuration is *idle* when all threads are *idle*.

The transition function $\rightarrow$ in Figure 5.6 is determined by a program P and event set E , and maps a configuration $c_1 \in C$ and thread identifier $k \in \mathbb{N}$ to another configuration c_2 and label $\lambda = \langle i, j, a \rangle$ where i and j are invocation and event identifiers, and $a \in A$ is an action — we write $\text{invoc}(\lambda)$, $\text{event}(\lambda)$, and $\text{act}(\lambda)$ to refer to i , j , and a , respectively. We write $q[u]$ and $t[u]$, resp., to denote the sets $q \cup \{u\}$ and $t \cup \{u\}$. EVENT transitions mark the beginnings of events. We assume that all events are initiated on thread 0, which is also referred to as the *main* thread. Also, for simplicity, we assume that events can be initiated arbitrarily at any time. Adding causality constraints

between events, e.g., one event can be initiated only when a certain action has been executed, is possible but tedious. CALL transitions create pending asynchronous invocations, DISPATCH transitions begin the execution of pending invocations, and RETURN transitions signal their end. END EVENT transitions mark the end of an event and by an abuse of notation, they map c_1 and k to a configuration c_2 and two labels, $\text{return}(i)$ denoting the end of the asynchronous invocation and $\text{end}(j)$ denoting the end of the event. All other transitions are LOCAL.

An *execution* of a program P with event set E to configuration c_n is a configuration sequence $c_0c_1 \dots c_n$ such that $c_m \xrightarrow{k_m, \lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We say that c_n is reachable in P with E under the weak semantics, and we call the sequence $\lambda_1 \dots \lambda_n$ the *trace* of $c_0c_1 \dots c_n$. The *reachable states* of P with E , denoted $R_w(P, E)$, is the set of global states in reachable idle configurations.

The set of traces of P with E under the weak semantics is denoted by $\llbracket P, E \rrbracket_w$. We may omit P when it is understood from the context, and write $\llbracket E \rrbracket_w$ instead of $\llbracket P, E \rrbracket_w$.

The *call tree* of a trace τ is a ranked tree $\text{CallTree}_\tau = \langle V, E, O \rangle$ where V are the invocation identifiers in τ , and the set of edges E contains an edge from i_1 to i_2 whenever i_2 is invoked by i_1 , i.e., τ contains a label $\langle i_1, -, \text{invoke}(i_2) \rangle$. The function $O : E \rightarrow \mathbb{N}$ labels each edge (i_1, i_2) with an integer n whenever i_2 is the n th invocation made by i_1 , i.e., $\langle i_1, -, \text{invoke}(i_2) \rangle$ is the n th label of the form $\langle i_1, -, \text{invoke}(-) \rangle$ occurring in τ (reading τ from left to right).

5.3.3 Strong Asynchronous Semantics

Conversely to the weak semantics of Section 5.3.2, our strong semantics minimizes the set of possible program behaviors by executing events independently, without interference from others, and by executing all asynchronous procedure invocations serially in a *fixed* order. We explain the intuition behind the choice of this order using the event `searchForNews` in Figure 5.4. A reasonable expectation when reading the code of this event is that the keyword is added to the search history only after the

fetching of the news containing that keyword succeeds. This expectation corresponds to executing the tasks according to the DFS traversal of the call tree. In general, this traversal is relevant because it preserves causality constraints which are imprinted in the structure of the code. Therefore, for every trace produced by our strong semantics, the asynchronous invocations start executing according to the DFS traversal of its call tree.

Configurations thus maintain a stack of queues of tasks, where a queue contains the tasks produced by a single invocation. Let U_0 be the set of tasks $u = \langle \ell, i, j, 0 \rangle$ executing on thread 0. Formally, a *configuration* $c = \langle g, u, q \rangle$ is a global state $g \in G$ along with a possibly-empty task placeholder $u \in (U_0 \cup \{\perp\})$ and a sequence of $q \in (\text{Tuples}(U_0))^*$ of tuples of tasks representing the store of pending asynchronous procedure invocations, where ε denotes the empty sequence. Let C_s denote the set of configurations. We say that c is *idle* when $u = \perp$.

The transition function $\Rightarrow$ representing the strong semantics is a restriction of $\rightarrow$ where the rules CALL, DISPATCH are redefined as in Figure 5.7. Every task created by the CALL transition is restricted to execute on the *main* thread and it is inserted at the end of the last tuple in q . For a tuple f , the tuple obtained by adding i_2 at the end of f is denoted by $\langle f, i_2 \rangle$. DISPATCH extracts a pending task from the beginning of the last tuple in q , and adds a new *empty* tuple to q (for storing the tasks created during the newly started invocation). The empty tuple is denoted by $\langle \rangle$. Moreover, the rules RETURN and END EVENT remove the last tuple in q if it is empty.

An *execution* of a program P with event set E to configuration c_n is a configuration sequence $c_0 c_1 \dots c_n$ s.t. $c_m \xrightarrow{0, \lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We say that c_n is *reachable* in P with E under the strong semantics, and we call the sequence $\lambda_1 \dots \lambda_n$ the *trace* of $c_0 c_1 \dots c_n$. The *reachable states* of P with E , denoted $R_s(P, E)$, is the set of global states reachable in idle configurations.

The set of traces of P with E under the strong semantics is denoted by $\llbracket P, E \rrbracket_s$ (P may be omitted).

$$\begin{array}{c}
 \text{CALL} \\
 \frac{P(g, \ell_1) = \langle -, \ell'_1, \text{invoke}(\ell_2, w) \rangle \quad u_2 = \langle \ell_2, i_2, j, 0 \rangle \quad i_2 \text{ is fresh}}{g, t[\langle \ell_1, i, j, k \rangle], q \cdot f \xrightarrow{0, \langle i, j, \text{invoke}(i_2) \rangle} g, t[\langle \ell'_1, i, j, k \rangle], q \cdot \langle f, i_2 \rangle} \\
 \\
 \text{DISPATCH} \\
 \frac{u = \langle \ell, i, j, k \rangle \quad 0 \text{ is idle} \quad f = \langle u, f' \rangle \quad q' \text{ is } \langle \rangle \text{ if } f' = \langle \rangle \text{ or } f' \cdot \langle \rangle, \text{ otherwise}}{g, \perp, q \cdot f \xrightarrow{0, \langle i, j, \text{begin}(i) \rangle} g, u, q \cdot q'}
 \end{array}$$

Figure 5.7: The strong transition function $\Rightarrow$ determined by a program P and event set E .

5.4 Robustness of Asynchronous Programs

Our robustness criterion is defined as the equality of the strong and weak semantics of a given program (§ 5.4.1), and it is decomposed into two independently-checkable criteria, event serializability and event determinism (§ 5.4.2).

5.4.1 Strong vs Weak Asynchronous Semantics

Given a program P with event set E , each execution under the strong semantics can be simulated by an execution under the weak semantics: the latter corresponds to a special scheduling policy that consists in executing all tasks created by an event before starting executing tasks corresponding to another event, and moreover, tasks are executed atomically, in the order given by the DFS traversal of the call tree. This implies that the weak semantics is a relaxation of the strong semantics, and therefore, $R_s(P, E) \subseteq R_w(P, E)$.

The reverse direction is the most interesting one:

Definition 5.4.1 (Robustness). *A program P with events E is robust against asynchrony (or simply robust) when all reachable states in the weak semantics are also reachable in the strong semantics: $R_w(P, E) \subseteq R_s(P, E)$.*

Robustness means that for the considered program, the concurrency introduced by the weak semantics does not modify the set of observable states, i.e., $R_w(P, E) =$

$R_s(P, E)$.

5.4.2 Event-Serializability and Event-Determinism

We introduce in this section two correctness criteria that capture precisely the notion of robustness against asynchrony.

We say an execution with trace $\lambda_1 \cdots \lambda_n$ is *event-serial* when for all $n_1 < n_3$, if $\text{act}(\lambda_{n_1}) = \text{start}(j)$ and $\text{act}(\lambda_{n_3}) = \text{start}(j')$, then there is n_2 such that $n_1 < n_2 < n_3$ and $\text{act}(\lambda_{n_2}) = \text{end}(j)$.

Definition 5.4.2 (Event-serializability). *A program P with events E is event-serializable if every global state in $R_w(P, E)$ can be reached by an event-serial execution.*

Our definition of serializability does not require that every state reachable by an execution of the weak semantics should also be reached by a serial execution that has the same set of events. This is different from the standard notion of serializability [Papadimitriou, 1979]. The latter can be achieved by adding a history variable to the weak semantics that records the set of executed events.

Given an event e , an *e-execution starting from global state g_0* is a g_0 -initialized execution (according to the weak semantics) with trace $\lambda_1 \cdots \lambda_n$ such that (1) $\text{act}(\lambda_1) = \text{start}(j)$, (2) $\text{act}(\lambda_n) = \text{end}(j)$, for some j , and (3) for every $m \in \mathbb{N}$ such that $1 < m < n$, $\text{act}(\lambda_m)$ is neither a start nor an end action. Intuitively, we consider executions of individual events, from their starting point until the completion of all the tasks they have created. Then, let $R_w(P, g_0, e)$ be the set of global states in final configurations of e -executions starting from g_0 . Notice that e -executions from g_0 differ by the scheduling order of the tasks created by e that are running in parallel on different threads.

Definition 5.4.3 (Event-determinism). *An event e of a program P is deterministic if for every global state g_0 , the set $R_w(P, g_0, e)$ is a singleton or empty. A program P is event-deterministic, if every event of P is deterministic.*

Notice that our notion of determinism is defined for events that are running alone, without interference of other events.

Theorem 4. *A program is robust against asynchrony if and only if it is event-serializable and event-deterministic.*

5.5 Conflict Robustness

Conflict based notion of serializability is introduced in [Papadimitriou, 1979] in the context of database transactions. Conflict serializability defines a *conflict relation* between the read and write actions of transactions without considering the data values read or written in the actions. While being more restrictive than serializability, conflict serializability is tractable to monitor.

Following the idea of using conflict based syntactic definitions introduced in the other contexts, we define the notion of *conflict robustness* that is the conjunction of two properties: *conflict-event serializability* and *conflict-event determinism*.

5.5.1 Conflict-Event Serializability.

Conflict relation. Let $\prec \subseteq A \times A$ be a *conflict relation* relates any two actions a, a' accessing the same variable, i.e., $a, a' \in \{read(x), write(x)\}$ for some x , one of them being a write. Formally,

$$a \prec a' \Leftrightarrow \exists x : [(a = read(x) \wedge a' = write(x)) \vee \\ (a = write(x) \wedge a' = read(x)) \vee \\ (a = write(x) \wedge a' = write(x))]$$

The conflicts given in the formal definition are called a “write after read” (WAR), “read after write” (RAW) or “write after read” (WAW) conflict respectively.

A trace is conflict-event serializable iff the “conflict-event graph” which tracks the conflict relation between concurrent event instances is acyclic.

Conflict-event graph. The *conflict-event graph* (or *precedence graph*) of a trace τ is the directed graph $EvG_\tau = \langle V, E \rangle$ whose nodes V are the event identifiers of τ , and which contains an edge from j_1 to j_2 when τ contains a pair of labels λ_1 and λ_2 such that λ_1 occurs before λ_2 , $act(\lambda_1) \prec act(\lambda_2)$, $event(\lambda_1) = j_1$, and $event(\lambda_2) = j_2$.

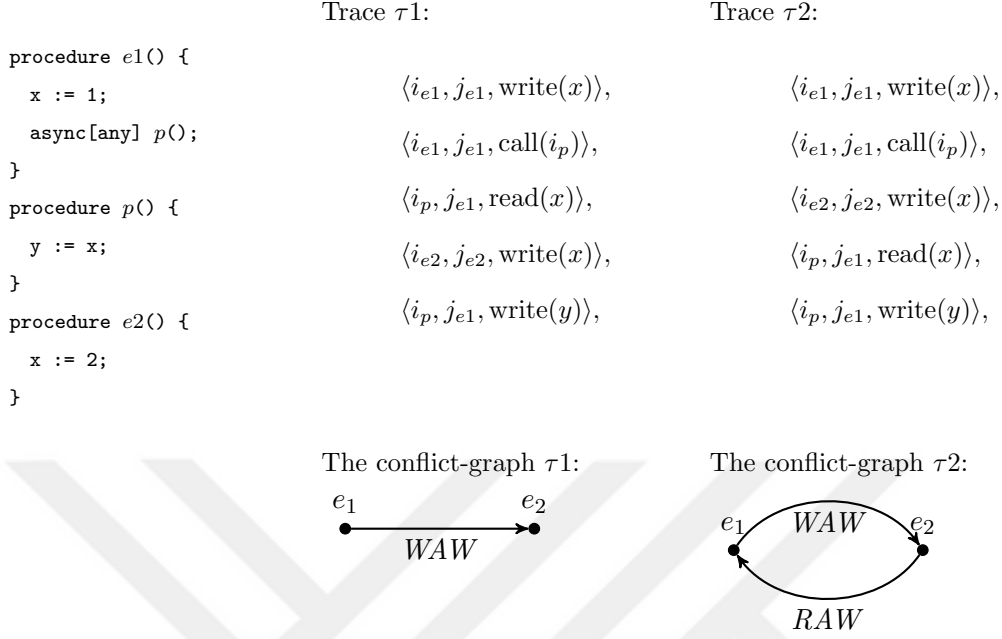


Figure 5.8: Execution traces τ_1 and τ_2 and their conflict-event graphs.

Intuitively, an edge from event j_1 to event j_2 in the conflict-event graph represents a precedence relation between the handlers of the events j_1 and j_2 . There exists an edge from j_1 to j_2 , if an action in j_2 's event handler is in conflict with an action executed in j_1 's event handler. An event serializable execution ensures all precedence relations to be in the same direction, hence the execution is conflict-equivalent to a serial execution of the handlers of the events $j_1; j_2$.

Definition 5.5.1. A trace τ is called *conflict-event serializable* when EvG_τ is acyclic. A program P with event set E is *conflict-event serializable* iff every trace in $\llbracket P, E \rrbracket_w$ is conflict serializable.

Figure 5.8 shows the conflict-event graphs of two traces of the execution of two event handler procedures $e1$ and $e2$. The execution trace τ_1 is conflict-serializable and it is equivalent to the execution of the event handlers serially in the order of $e1; e2$. The conflict-event graph of the trace τ_2 is cyclic and it is not conflict serializable.

Conflict-preserving permutation. A permutation τ' of a trace τ is *conflict-preserving* when every pair λ_1, λ_2 of labels in τ appear in the same order in τ'

whenever $\text{act}(\lambda_1) \prec \text{act}(\lambda_2)$. Note that a conflict-preserving permutation τ' leads to the same global state as the original trace τ . In the text, when we use permutation we mean conflict-preserving permutation. A trace τ is conflict-event serializable iff it is a conflict-preserving permutation of an event-serial trace. The conflict-serializable execution trace τ_1 in Figure 5.8 is a permutation of the serial execution of the event handlers $e_1; e_2$, as it preserves the order of conflicting actions in the serial execution.

Conflict serializability ensures serializability since different orderings of non-conflicting actions does not change the read or final state of the data values.

Theorem 5. *A program P with event set E is event-serializable when it is conflict-event serializable.*

The work by [Farzan and Madhusudan, 2008] studies the algorithmics of monitoring atomicity of concurrent programs in the context of multithreaded programs. For a fixed number of threads, they prove that the conflict-serializability problem is PSPACE-complete [Alur et al., 2000, Farzan and Madhusudan, 2008] by showing that it is sufficient to track a bounded summary of the conflict-graph. For an unbounded number of threads (as in our case), the problem is proven to be EXPSPACE-complete [Bouajjani et al., 2013].

Theorem 6. *Checking conflict-serializability of a bounded-data program is EXPSPACE-complete.*

Event conflict-serializability of an execution trace can be monitored by maintaining an event conflict graph of the accesses of each event in execution. An event-driven asynchronous program can be instrumented to maintain its conflict-graph of an execution and to reach an error location iff the conflict-graph is acyclic.

In Section 6.3.1, we present a conflict-event serializability checker for Android applications based on the work by [Alur et al., 2000, Farzan and Madhusudan, 2008]. It is not a tool for the verification of conflict-event serializability but a violation detection tool that checks whether an analyzed execution trace is event conflict-serializable or not. Although the analysis coverage can be increased by employing

some heuristics to collect a more diverse set of execution traces, it is hard to cover all possible interleavings and ensure conflict-event serializability.

We provide a novel algorithmic method to *verify* event conflict-serializability of a program in Section 6.3.2.2. Given an event-driven asynchronous program and a sequence of event handler invocations, it decides whether all possible execution traces is event conflict-serializability or not. Remarkably, although the program under analysis is concurrent and has unbounded number of asynchronous tasks, the complexity of our verification method reduces to the reachability analysis in sequential programs.

5.5.2 Conflict Determinism.

We define *conflict determinism*, which is also based on the acyclicity of a certain class of “conflict graphs”, called *conflict-invocation graphs*. These graphs represent the conflicts between the asynchronous invocations, but also the order in which these invocations would be executed under the strong semantics, i.e., the DFS traversal of the call tree.

Recall that an event e is called deterministic iff every e -trace (the trace of an e -execution) starting from some fixed configuration c_0 leads to the same global state, or equivalently, iff given a fixed e -trace t_0 from c_0 where asynchronous procedure invocations are executed atomically, every other e -trace τ from c_0 leads to the same global state as t_0 . If the conflict-invocation graph of every trace τ of an event e is acyclic, then e is deterministic because every trace τ is a conflict-preserving permutation of the trace t_0 corresponding to the strong semantics, and thus leads to the same global state as t_0 .

Conflict-invocation graph Given a trace τ , let $<_{dfs}$ be the total order between the invocation identifiers in τ defined by the DFS traversal of $CallTree_\tau$. The *conflict-invocation graph* of a trace τ is the directed graph $InvG(\tau) = \langle V, E \rangle$ whose nodes V are the asynchronous invocation identifiers in τ , and which contains an edge from i_1 to i_2 when $i_1 <_{dfs} i_2$, or τ contains a pair of labels λ_1 and λ_2 of i_1 and i_2 , resp., such that $act(\lambda_1) \prec act(\lambda_2)$ and λ_1 occurs before λ_2 .

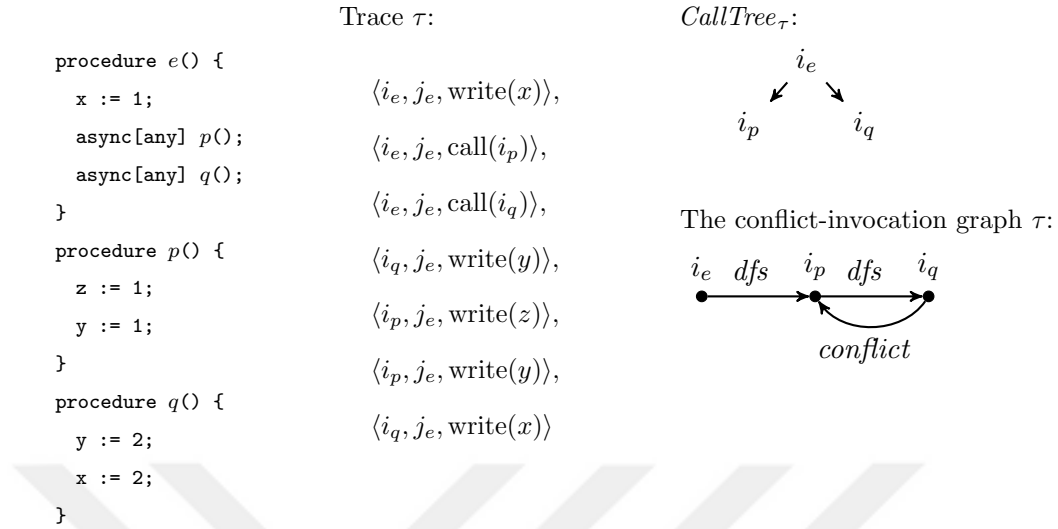


Figure 5.9: An event e , an e -trace τ and the conflict-invocation graph of τ .

Figure 5.9 shows the conflict-invocation graph of a trace τ of an event handler procedure e .

Definition 5.5.2. A trace τ is *DFS-serial* iff $\text{InvG}(\tau)$ is acyclic. An event e is *conflict-deterministic* iff every trace in $\llbracket e \rrbracket_w$ is *DFS-serial*.

A trace τ is called *invocation-serial* iff for every three labels $\lambda_1, \lambda_2, \lambda_3$ occurring in τ in this order, if $\text{invoc}(\lambda_1) = \text{invoc}(\lambda_3)$, then $\text{invoc}(\lambda_1) = \text{invoc}(\lambda_2)$. For an event e , a DFS-serial trace τ in $\llbracket e \rrbracket_w$ is a permutation of an invocation-serial trace $\tau_0 \in \llbracket e \rrbracket_w$ where $\text{invoc}(\lambda_1) <_{\text{dfs}} \text{invoc}(\lambda_2)$ for every two labels λ_1 and λ_2 occurring in this order in τ_0 .

For instance, the conflict-invocation graph in Figure 5.9. contains a cycle and thus, the considered trace is not DFS-serial. One can obtain a DFS-serial trace if the $\text{write}(x)$ action in the asynchronous procedures p takes place before the $\text{write}(x)$ action in q .

Theorem 7. An event is deterministic when it is conflict-deterministic.

Chapter 6

VERIFYING ROBUSTNESS AGAINST ASYNCHRONY ¹

This chapter presents a novel algorithmic approach for checking robustness against asynchrony as a conjunction of event-serializability and event-determinism. To search efficiently for event-serializability and event-determinism violations, we make use of *conflict-based approximations* in the style of [Papadimitriou, 1979]. Indeed, these conflict-based criteria do not take into account actual data values, but rather syntactical dependencies between operations, e.g., writing to the same variable, which makes them stronger, but also “easier” to check, while still accurate enough for catching real bugs, introducing rarely false positives, as our experiments show. We reduce verifying conflict event-serializability and conflict event-determinism to detecting cycles in appropriately defined dependency (or happen-before) relations between concurrent events and asynchronous procedure invocations, respectively. Our key contribution is that, surprisingly, these cycle detections can be done by reasoning about the computations of *sequential programs* instead of concurrent programs, avoiding explicit encodings of (potentially unbounded) sets of pending tasks, and exploring all their possible interleavings. Let us explain this in more details.

First, an event handler is conflict-deterministic if all its executions have conflict-preserving counterparts where tasks are scheduled in the DFS order of the call-graph of tasks (i.e., corresponding to an execution under the strong semantics). Then, we show that detecting a conflict-determinism violation, i.e., an asynchronous execution with no DFS counterpart, can be done by reasoning about an instrumented version of the procedural program obtained from the code of the event handler by turning asynchronous calls to synchronous ones. The instrumentation simulates, by

¹The work in this chapter is a joint work with Ahmed Bouajjani, Constantin Enea, Michael Emmi and Serdar Tasiran.

nondeterministically skipping tasks, the delaying of tasks done by the asynchronous program, and maintains information on the dependencies between nondeterministically guessed actions that cannot be executed in DFS order. We show that the number of actions to track is finite (and small) because, and this is crucial, only a special type of conflict-determinism violations (i.e., executions with a conflict cycle), called border-line violations, must be considered. Therefore, we obtain that checking conflict-determinism of an event handler can be reduced *polynomially* to a state reachability problem in a *sequential program* (assuming a fixed number of variables and procedures).

It is known that checking conflict-serializability is hard in general: assuming a fixed data domain and absence of recursive procedure calls, the problem is PSPACE-complete for a fixed number of threads [Alur et al., 2000, Farzan and Madhusudan, 2008], and EXPSPACE-complete for an unbounded number of threads [Bouajjani et al., 2013]. However, we prove that, in the setting we consider in this work, the problem of checking conflict-robustness is polynomial! In fact, we show that when events are conflict-deterministic, the problem of checking conflict-serializability can be reduced to a reachability problem in a sequential program. Again, we prove that it is sufficient to focus on a particular class of (border-line) violations of conflict-serializability. Then, we show that detecting these violations can be done by reasoning about the executions of a program where events are executed in a sequential manner, in any order (chosen nondeterministically), and where the tasks generated by each event are executed in DFS order. For that, we define an instrumentation of that program that consists in simulating the delaying effects of the weak semantics, guessing the actions involved in the violation and tracking the dependencies between them in order to check the correctness of the guess (that they indeed form a cycle). The cycle detection in the case of conflict-serializability is technically more complex than in the case of conflict-determinism. But still, a crucial point in the reduction is that we do not need to store the whole cycle during the search, but it is enough to maintain a fixed number of variables to traverse the elements of this cycle. This leads to a polynomial reduction of the conflict-robustness problem to a reachability problem in a sequential program.

Our reductions hold regardless of the used data domain, for programs with recursive procedure calls, and unbounded numbers of events and tasks. These reductions allow to leverage existing analysis tools for sequential programs to check conflict-robustness. When the data domain is bounded, we obtain a polynomial-time algorithm for checking conflict-robustness for event-driven asynchronous programs (with recursive procedure calls, and unboundedly many events and tasks).

We validate our approach on a set of real-life applications, showing that all detected robustness violations are undesirable behaviours. Interestingly, the use of conflict versions of the correctness criteria characterizing robustness is efficient and quite accurate, producing only few false positives (that can be eliminated easily).

In fact, robustness violations cause safety violations that are due to concurrency. Our work shows that checking absence of concurrency bugs in for event-driven asynchronous programs can be done without explicit consideration of all their concurrent computations. This avoids state explosion since it allows to separate the issue of checking absence of concurrency bugs, from the issue of checking absence of safety bugs that are present even in the sequential version of the program. When a program is checked to be robust, which can be done using the strong (sequential) semantics, then checking a safety property (e.g. that an invariant holds) can be done efficiently again by considering only the strong semantics of the program.

6.1 Checking Conflict Determinism

We reduce the problem of checking conflict determinism of an event to a reachability problem in a *sequential* program. We present the reduction in two steps. First, conflict determinism of an event interpreted under the weak semantics, whose asynchronous invocations run concurrently, is reduced to a reachability problem in a program running on the strong semantics, where asynchronous invocations are executed serially (Section 6.1.1 and 6.1.2). The latter is then reduced to a reachability problem in a sequential program (Section 6.1.3).

This reduction uses the fact that a certain class of conflict determinism violations

can be simulated by a sequential program up to conflict-preserving permutations of actions (note that any conflict-preserving permutation of a violation is also a violation). This class of violations called *borderline violations* are minimal in the sense that removing the last action leads to a correct trace. Besides the simulation, we show that fixed-size additional memory is required to witness the conflicts inducing a cycle in the conflict graph.

Definition 6.1.1 (Borderline Conflict Determinism Violation). *A trace τ is a borderline violation to conflict determinism if it is not DFS-serial but every strict prefix of τ is DFS-serial.*

For instance, the trace τ_1 given in Figure 6.1(a) contains a borderline violation. This trace is generated by an event e that invokes two procedures p and q in this order, each procedure on a different thread. The only conflict between memory accesses is that between the $wr(x)$ actions in q and resp., p . The conflict-invocation graph of τ_1 contains a cycle between the invocations of p and q : the edge from the invocation of p to that of q is implied by the fact that p is invoked before q within the same procedure (we have “ $p <_{dfs} q$ ”), and the edge in the other direction exists because q writes to the variable x before p does. The trace τ_1 until after the second $wr(x)$ is a borderline violation since its maximal strict prefix (without the second $wr(x)$) is DFS-serial. The last label of a borderline violation τ , in this example $wr(x)$, is called the *pivot* of τ . The label of τ which precedes and conflicts with its pivot and which induces the cycle in its conflict-invocation graph is called the *root* of τ . Formally, if i_1 is the invocation containing the pivot of τ , the root of τ is an action conflicting with the pivot and which is included in an invocation i_2 such that $i_1 <_{dfs} i_2$. For the trace in Figure 6.1(a), the root is the action $wr(x)$ in the invocation of q .

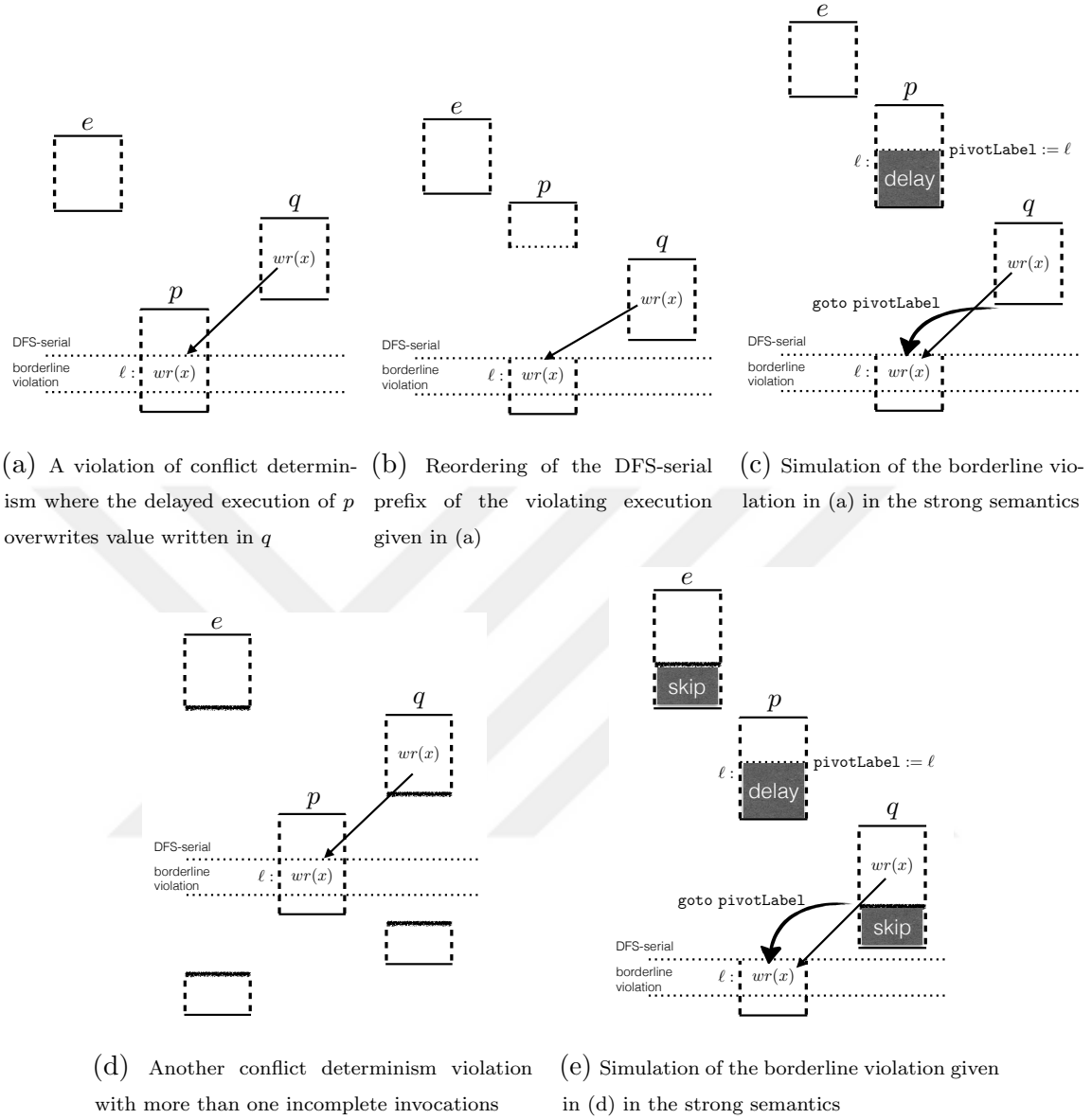


Figure 6.1: Simulating borderline conflict determinism violations on the strong semantics. The event e makes two fresh thread asynchronous invocations to p and q in this order. Boxes represent sequences of trace labels ordered from top to bottom. Actions of the same thread are aligned vertically. The arrows represents transition label conflicts. For readability, we omit the event and task identifiers in the trace labels and keep only the memory accesses. The grey blocks labeled by *delay*, resp., *skip*, denote sequences of actions that are delayed, resp., skipped.

6.1.1 Simulating Borderline Violations

We define a code-to-code translation from an event e to an event $\text{detStr}^-(e)$ which simulates² permutations of every DFS-serial or borderline violation trace in $\llbracket e \rrbracket_w$. The event $\text{detStr}^-(e)$ uses additional non-deterministically enabled statements to simulate the particular interleavings present in those traces. The instrumentation required to witness violations is introduced in Section 6.1.2.d

6.1.1.1 Overview

We give an informal description of the translation using as examples the traces pictured in Figure 6.1 and 6.2.

Delaying the pivot We first explain the simulation of the invocation that contains the pivot, which may interfere with invocations that are supposed to be executed *later* under the strong semantics.

For the borderline violation in Figure 6.1(a), the invocation of p that contains the pivot $wr(x)$ destroys the value written to x by q , an invocation which is executed after p under the strong semantics.

The maximal strict prefix (ending before the second $wr(x)$) is DFS-serial and can be reordered to a trace where the order between transition labels is consistent with the invocation order (i.e., e before p and before q). Figure 6.1(b) pictures such a reordering, denoted by τ'_1 .

Our goal is to show that the trace τ'_1 can be simulated by an execution under the strong semantics of a slightly modified version of e . First note that τ'_1 is not admitted by the strong semantics of e because the invocation of p is only *partially* included in this prefix. And the strong semantics executes every task until completion. However, it is possible to “delay” the execution of the pivot $wr(x)$ in p until q finishes, even under the strong semantics, by adding a suitable set of auxiliary variables to e . This

²We refer to the standard notion of (stuttering) simulation where (sequences of) transitions in $\text{detStr}^-(e)$ are mapped to transitions of e .

mechanism is pictured in Figure 6.1(c) in which the grey block labeled by `delay` denotes sequences of actions that are delayed. Every statement in the procedure p is guarded by (the negation of) an auxiliary boolean flag `skip` which can be *non-deterministically* flipped to `true` in order to skip over statements. Moreover, an auxiliary global variable `pivotLabel` will record the next control flow label ℓ when this flag is set to `true`. Then, extending the invocation of q with `goto pivotLabel` allows to resume the invocation of p and execute the pivot. To simulate every borderline violation, the `goto` statement is non-deterministically enabled in every invocation.

Incomplete invocations While the violation in Figure 6.1(a) includes only one incomplete invocation (the one containing the pivot) this is always the case. A borderline violation may contain unboundedly-many other incomplete invocations. For instance, the violation in Figure 6.1(d) includes incomplete invocations of e and q (they finish after the pivot). Should the simulation of this borderline violation execute e and q entirely, the pivot may never be enabled. The correct simulation, pictured in Figure 6.1(e), will make use of the same mechanism based on the boolean flag `skip` in order to skip over statements in e and q . In general, an invocation can be skipped in its entirety. This simulation also shows that the `goto` statement can be executed after an incomplete invocation.

Main thread invocations The last issue concerns the main thread which has the particularity of being able to execute more than one invocation (all the other threads execute a single invocation). He executes invocations serially and only the last one may be incomplete. For instance, consider the DFS-serial trace τ_3 pictured in Figure 6.2(a). The call tree of this trace is given in the lower part of the same figure. Since p_1 is invoked before p_3 , a DFS-serial permutation τ'_3 of τ_3 contains the *incomplete* invocation of p_1 before the *complete* invocation of p_3 , as shown in Figure 6.2(b). None of the semantics we defined allows such traces. The problem is that both invocations are executed by the main thread which has to complete a task before executing another one. Our simulation will however admit such traces but it will verify that they are

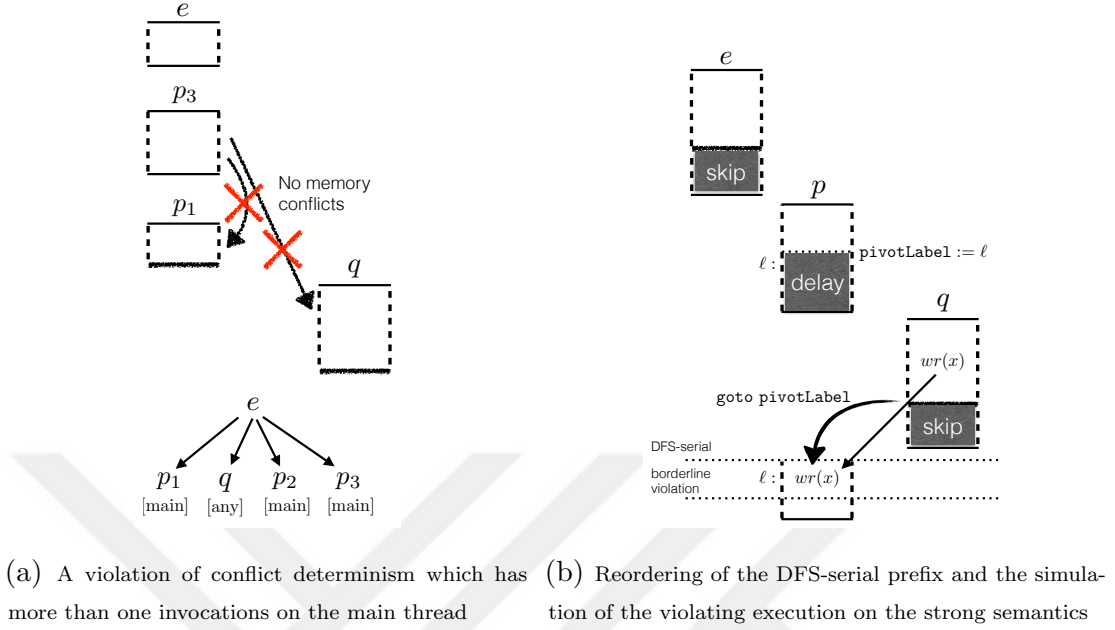


Figure 6.2: Simulating a DFS-serial trace on the strong semantics. The procedures p_1 , p_2 , and p_3 are executed on the main thread while q is executed on a different thread.

conflict-preserving permutations of valid traces. This verification procedure (included in the definition of $\text{detStr}^-(e)$) checks that the conflict invocation graph doesn't contain a path of *memory conflicts*, i.e., conflicts induced by read and write accesses, from the incomplete invocation on the main thread to any future complete invocation on the same thread. Let us consider again the trace τ_3 in Figure 6.1(d). Since τ_3 is DFS-serial, its conflict invocation graph doesn't contain paths of memory conflicts from p_3 to any other invocation ordered before p_3 in the DFS traversal of the call tree. This includes the incomplete invocation p_1 and q . For the permutation τ'_3 , this implies that its conflict invocation graph contains no paths of memory conflicts from p_1 to p_3 . When a trace satisfies this condition, i.e., an incomplete invocation on the main thread doesn't conflict with a future complete invocation on the same thread, all the complete invocations on the main thread can be reordered before the incomplete one (preserving the order between conflicting trace labels) and this results in a valid trace

(under the weak semantics). The simulation of τ'_3 on the strong semantics, pictured in Figure 6.2(b), enables this verification procedure during the invocation of p_1 because it is executed on the main thread and it skips over statements. It is also possible that other invocations on the main thread, e.g., p_2 , are skipped *entirely*. This doesn't pose any particular semantic issue.

6.1.1.2 Notations

We introduce several notations used in the definition of $\text{detStr}^-(e)$. This event is obtained by rewriting every statement s of a procedure transitively invoked by e to a code fragment $s_1; \text{if}(c) \text{ then } s; s_2$ where s_1 and s_2 are statements and c is a boolean expression. We use $\text{before}(s)$, $\text{guard}(s)$, and $\text{after}(s)$ to refer to s_1 , c , and s_2 , respectively. For every statement s , $\ell(s)$ denotes the control flow label of s , that can be used for instance in goto statements. Also, $\text{rdSet}(s)$, resp., $\text{wrSet}(s)$, is the set of global variables read, resp., written, by s . We have $\text{wrSet}(s) = \{x\}$ and $\text{rdSet}(s) = \emptyset$ when s is $x := y$, and $\text{wrSet}(s) = \emptyset$ and $\text{rdSet}(s) = \{x\}$ when s is $y := x$. Otherwise, $\text{rdSet}(s) = \text{wrSet}(s) = \emptyset$.

We assume that every procedure p is augmented with two local variables **rdSetProc** and **wrSetProc** tracking the global variables read and written by p , respectively ($\text{rdSet}(s)$ and $\text{wrSet}(s)$ are added to **rdSetProc** and **wrSetProc**, respectively, after every statement s that gets executed).

The instrumentation uses the non-deterministic choice denoted by $*$ (formally, $*$ is a distinguished boolean variable that evaluates non-deterministically to **true** or **false**). To refer to the different non-deterministic choices in the instrumentation, we may index them with natural numbers, but this has no impact on their semantics.

To reduce clutter in the instrumentation, we use $[s]_{\text{ev}}(\mathbf{b})$ to denote a statement s that is executed at most once during the execution of the event and the boolean variable $\mathbf{b}$ is set to **true** when s gets executed.

For an event e , let $\mathcal{P}(e)$ be the set of the procedures possibly invoked by e , which is defined inductively by: (1) $e \in \mathcal{P}(e)$ and (2) for every $p \in \mathcal{P}(e)$, if $\text{async}[w] \ q(\mathbf{y})$

occurs syntactically in the code of e , then $q \in \mathcal{P}(e)$. Also, let $\mathcal{P}_0(e)$ be the subset of $\mathcal{P}(e)$ consisting of procedures posted to the main thread, i.e., in the previous inductive definition, we take $w = \text{main}$. W.l.o.g. we assume that the procedures in $\mathcal{P}_0(e)$ are distinct from the procedures q contained in asynchronous invocations of the form “`async[any]  $q(\dots)$` ” executed on other threads.

All the boolean variables added by the instrumentation are initially set to `false`.

6.1.1.3 Defining the instrumentation

Dealing with fresh thread invocations To simulate incomplete invocations executed by threads different from the main thread, every procedure in $\mathcal{P}(e) \setminus \mathcal{P}_0(e)$ is augmented with a boolean flag `skip` that is non-deterministically set to `true`. When `skip` is set to `true`, the rest of statements are skipped and the first skipped statement *may* be chosen as the pivot and its label stored in `pivotLabel`. The pivot may get executed non-deterministically at a later time.

Formally, for every statement s of procedure $p \in \mathcal{P}(e) \setminus \mathcal{P}_0(e)$, $\text{guard}(s) = \neg \text{skip}$ and $\text{before}(s)$ is defined by:

```

if ( !skip & *1 ) then
    skip := true
    if ( *2 ) then
        [pivotLabel :=  $\ell(s)$ ]ev(pivotSet)
    if (skip & pivotSet & *3) then
        [goto pivotLabel]ev(gotoDone)

```

where `skip` is a local variable and `pivotLabel` is a global variable.

Dealing with main thread invocations For procedures in $\mathcal{P}_0(e)$, the instrumentation ensures that at most one invocation of such a procedure is incomplete, and also, that the conflict invocation graph contains no path of memory conflicts from such an incomplete invocation to any future complete invocation of a procedure in $\mathcal{P}_0(e)$. Such paths of memory conflicts may cross invocations of procedures which are not in

$\mathcal{P}_0(e)$, therefore the instrumentation of the latter must also be modified.

To simulate an incomplete invocation on the main thread, for every statement s of a procedure $p \in \mathcal{P}_0(e)$, $\text{before}(s)$ is set to:

```

if ( *4 ) then
  [skip := true]ev(skipMainSet)
  rdSetGlobal := rdSetProc; wrSetGlobal := wrSetProc;
  if ( *5 ) then
    [pivotLabel :=  $\ell(s)$ ]ev(pivotSet)

```

where **skip** is a boolean local variable. As for invocations executed on other threads, the first skipped statement may be chosen as the pivot. To be able to track paths of memory conflicts, the variables read and written during the incomplete invocation are stored in the global variables **rdSetGlobal** and **wrSetGlobal**, respectively. For invocations of procedures $p \in \mathcal{P}_0(e)$, **skip** can be set to **true** at most once during the execution of the event.

Other tasks posted to the main thread can be skipped entirely or executed completely, by setting a local flag **skipProc**. When they are executed completely, a global boolean flag **validMain** is used to witness that they are not the destination of a path of memory conflicts as explained above. At the beginning of each procedure, **validMain** is reset to **false**. Thus, the following is added at the beginning of every $p \in \mathcal{P}_0(e)$:

```

if ( *6 ) then
  skipProc := true;
  validMain := false;

```

Then, for every statement s of a procedure $p \in \mathcal{P}_0(e)$, $\text{guard}(s)$ is set to $\neg \text{skipProc} \ \& \ \neg \text{skip}$.

Once an incomplete invocation on the main thread is present, i.e., **skipMainSet** is **true**, the procedure for checking the absence of paths of memory accesses is enabled. For every statement s of every procedure $p \in \mathcal{P}(e)$, $\text{after}(s)$ is set to:

```

if ( skipMainSet & ( rdSetGlobal  $\cap$  wrSetProc  $\neq \emptyset$ 
                    | wrSetGlobal  $\cap$  rdSetProc  $\neq \emptyset$ 

```

```

| wrSetGlobal  $\cap$  wrSetProc  $\neq \emptyset$ 
| conflictDetected) ) then
rdSetGlobal := rdSetGlobal  $\cup$  rdSetProc;
wrSetGlobal := wrSetGlobal  $\cup$  wrSetProc;
conflictDetected := true;

```

where `conflictDetected` is a boolean local variable.

This conditional checks whether the current procedure conflicts with the incomplete invocation or transitively, with all the other invocations that conflict with the latter. If this is the case, then its set of memory accesses is continuously added to the global sets `rdSetGlobal` and `wrSetGlobal` of memory accesses.

When a main thread invocation finishes, if it has been executed completely and if it follows an incomplete main thread invocation, the instrumentation checks for absence of paths of memory conflicts and may non-deterministically choose to execute the pivot. Therefore, the following is added at the end of every $p \in \mathcal{P}_0(e)$:

```

if ( skipMainSet &  $\neg$  skipProc &  $\neg$  skip ) then
  assume  $\neg$  conflictDetected;
  validMain := true;
if ( pivotSet & *7 ) then
  [goto pivotLabel]ev(gotoDone)

```

Relationship between e and $\text{detStr}^-(e)$ The following result expresses the relationship between the original event e and $\text{detStr}^-(e)$. It shows that the strong semantics of $\text{detStr}^-(e)$ simulates permutations of all the DFS-serial traces and border-line violations of e under the weak semantics (modulo a thread id renaming). Moreover, every trace of $\text{detStr}^-(e)$ under the strong semantics where the last value of `validMain` is `true`, this set of traces being denoted by $\llbracket \text{detStr}^-(e) \rrbracket_s^{\text{validMain}}$, corresponds to a trace of e under the weak semantics (modulo the instrumentation added in $\text{detStr}^-(e)$ and a thread id renaming). For a trace t of $\text{detStr}^-(e)$, $\bar{t}$ is the trace obtained from t by erasing all transition labels corresponding to statements added by the instrumentation. For readability, we ignore the issue of renaming thread ids.

Theorem 8. *For every trace τ_1 in $\llbracket e \rrbracket_w$, if τ_1 is DFS-serial or a borderline conflict determinism violation, then there exists a trace τ_2 in $\llbracket \text{detStr}^-(e) \rrbracket_s$ such that $\tau_1' = \overline{\tau_2}$ is a conflict-preserving permutation of τ_1 . Moreover, for every trace τ_1 in $\llbracket \text{detStr}^-(e) \rrbracket_s^{\text{validMain}}$ there exists a trace τ_2 in $\llbracket e \rrbracket_w$ such that $\tau_2 = \overline{\tau_1}$.*

6.1.2 Witnessing Borderline Violations

The instrumentation used to verify that a trace is indeed a borderline violation consists in guessing a candidate for the root and then, when the pivot gets executed, checking whether it conflicts with the chosen candidate. For instance, if we consider the strong semantics simulation in Figure 6.1(c), the action $wr(x)$ in q is guessed as the root and its label is stored in an auxiliary variable `rootLabel`. This label is used to check that the root candidate conflicts with the pivot when the latter is executed. The root must be chosen after the pivot in order to guarantee that this leads to a cycle in the conflict invocation graph (i.e., the DFS traversal of the call tree orders the invocation containing the pivot before the one containing the root).

We define a new event $\text{detStr}(e)$ that sets an `error` flag to `true` whenever the current trace is not DFS serial and the root and pivot candidates are valid. This event is obtained from $\text{detStr}^-(e)$ by adding two global variables `error` and `rootLabel`, and

- concatenating the following to `before(s)`

```
if (!skip & pivotSet & *) then
  [rootLabel :=  $\ell(s)$ ]ev(rootSet)
```

This allows to non-deterministically choose s to be the root of the violation. In order to avoid choosing the pivot after the root, we must also replace `*2` and `*5` in $\text{detStr}^-(e)$ with `$\neg$  rootSet & *2` and `$\neg$  rootSet & *5`, respectively.

- concatenating the following to `after(s)`

```
if ( conflict(pivotLabel, rootLabel) & pivotLabel ==  $\ell(s)$  & gotoDone & rootSet )
  then
    error := true;
```

where $instrumentconflict(pivotLabel, rootLabel)$ is defined by

$$\begin{aligned} & rdSet(\ell^{-1}(pivotLabel)) \cap wrSet(\ell^{-1}(rootLabel)) \neq \emptyset \\ & | \quad rdSet(\ell^{-1}(rootLabel)) \cap wrSet(\ell^{-1}(pivotLabel)) \neq \emptyset \\ & | \quad wrSet(\ell^{-1}(rootLabel)) \cap wrSet(\ell^{-1}(pivotLabel)) \neq \emptyset \end{aligned}$$

This allows to validate that the root does indeed conflict with the pivot, right after the latter gets executed. If the conflict is validated, then `error` is set to `true`.

Since the added instrumentation only reads variables of $\detStr^-(e)$, the new event $\detStr(e)$ still satisfies the claim in Theorem 8.

Theorem 9. *An event e (under the weak semantics) satisfies conflict determinism iff the program $\detStr(e)$ under the strong semantics does not reach a state where `error` = `true`.*

Concerning complexity, $\detStr(e)$ can be constructed in linear time and its number of variables increases linearly in the number of variables and procedures of e .

6.1.3 Reduction to the Procedural Semantics

As a continuation to Theorem 9, we define a code-to-code translation from an event e to a *sequential* event $\text{seq}(e)$ such that $\text{seq}(e)$ admits exactly the set of traces of e under the strong semantics³.

Strong semantics vs procedural semantics Essentially, $\text{seq}(e)$ is obtained from e by rewriting asynchronous procedure invocations to regular procedure calls. However, this rewriting can't be applied directly because of the following issue. Consider a procedure p invoking another procedure q . If the invocation of q is asynchronous, the strong semantics executes p completely before starting q . Under the procedural

³Modulo the omission of the labels $\text{invoke}(i)$, $\text{begin}(i)$, $\text{return}(i)$ related to asynchronous invocations.

semantics, when q is invoked using a regular procedure call, the execution of p is blocked when q is invoked and resumed when q is completed. For instance, consider the event:

<pre> procedure e₁() { y := 1; async[main] p(); y := 2; } </pre>	<pre> procedure p() { y := 3; } </pre>
---	--

Executing e_1 on the strong semantics, we get the sequence of assignments $y := 1$, $y := 2$, $y := 3$. Rewriting `async[main] p()` to a regular procedure call `call p()`, we get an event that executes $y := 1$, $y := 3$, $y := 2$ in this order.

This issue doesn't exist for the class of events where all the asynchronous invocations occur at the end of the procedures. For instance consider the following event e_2 :

<pre> procedure e₂() { x := 1; async[main] p(); } </pre>	<pre> procedure p() { x := 2; async[main] q(); } </pre>	<pre> procedure p() { x := 3; } </pre>
---	---	--

Rewriting every `async[main] _` to a procedure call `call _`, we get an event that executes the assignments on x in exactly the same order as e_2 under the strong semantics. This holds because the strong semantics executes the asynchronous invocations according to the DFS traversal of the call tree, which corresponds to the “stack” semantics of procedure calls.

Therefore, the event $\text{seq}(e)$ is obtained in two steps. A first translation is used to move all asynchronous invocations at the end of the procedures. This results in an event having exactly the same strong semantics as the original one. Then, we replace every asynchronous invocation with a procedure call.

Defining $\text{seq}(e)$ The event e is extended with auxiliary data structures that store the names and the inputs of the asynchronous invocations. Using these data structures, all the invocations are delayed till the end of the encompassing procedure. Thus,

- each procedure p is extended with an auxiliary local variable `invocList` which stores a list of procedure names and inputs,
- when an asynchronous procedure q is invoked in p with inputs $\mathbf{y}$, the procedure name q together with its parameters $\mathbf{y}$ is appended to the local variable `invocList` of p without invoking q ,
- before returning from a procedure p , all the procedures stored in `invocList` are invoked in the order they are recorded.

For the event e_1 , this boils down to simply moving the invocation in e_1 at the end (i.e., after $\mathbf{y} := 1$). It is easy to see that the obtained event has the same strong semantics as the original event.

Let $\text{seq}(e)$ be the event obtained from e by applying the transformation above and then, replacing every asynchronous invocation `async` $[w]$ $p(\mathbf{y})$ with `call` $p(\mathbf{y})$.

For an event e , we overload the equality relation between traces $\tau_1 \in \llbracket e \rrbracket_s$ and $\tau_2 \in \llbracket \text{seq}(e) \rrbracket_s$ as follows: $\tau_1 = \tau_2$ iff removing the labels `invoke` (i) , `begin` (i) , `return` (i) with $i \in \mathbb{N}$ from τ_1 , and the transition labels corresponding to statements added by the instrumentation from τ_2 , we get the same trace.

A sequential program **Seq** has the same set of traces under the weak and the strong semantics, so its set of traces is denoted $\llbracket \text{Seq} \rrbracket$.

Theorem 10. *For any event e , $\llbracket e \rrbracket_s = \llbracket \text{seq}(e) \rrbracket$.*

For an event e , let $\text{detSeq}(e) = \text{seq}(\text{detStr}(e))$. By Theorem 10, $\text{detSeq}(e)$ still satisfies the claim in Theorem 8. The following is a direct consequence of Theorem 9 and 10.

Corollary 1. *An event e (under the weak semantics) satisfies conflict determinism iff the sequential event $\text{detSeq}(e)$ does not reach a state where `error` = `true`.*

Concerning complexity, let e be an event where each procedure invokes at most k other procedures, for some fixed k . Then, the time complexity of constructing

$\text{detSeq}(e)$ and its number of variables are quadratic in the number of variables and procedures of e and k .

6.2 Checking Conflict Robustness

Building on the reduction of conflict determinism to reachability in sequential programs, we show that a similar reduction can be obtained for conflict robustness. This reduction is based on two facts: (1) incomplete executions of conflict-deterministic events can be simulated by a sequential program, which has been proved in Section 6.1, and (2) conflict serializability for a set of conflict deterministic events can be again reduced to reachability in sequential programs. To prove the latter we use the concept of borderline violation, this time for conflict serializability. We show that interleavings corresponding to such violations can be simulated by a sequential program. This program behaves like a “most-general client” of the event-based program in the sense that it executes an arbitrary set of events, in an arbitrary order, but serially without interference from others. We show that the memory required to track the conflicts which induce a cycle in the conflict graph is of *bounded* size, although the conflict graph cycles are of unbounded size in general.

Definition 6.2.1 (Borderline Conflict Serializability Violation). *A trace t is a borderline violation to conflict serializability if it is not conflict serializable but every strict prefix of τ is conflict serializable.*

The trace τ_1 in Figure 6.3(a) contains a borderline violation. Its conflict-event graph contains a cycle between the three events e_1 , e_2 , and e_3 . The prefix of τ_1 ending just before $rd(z)$ satisfies conflict serializability. The last label of a borderline violation τ is called the *pivot* of τ (in this example $rd(z)$) and the event that contains the pivot is called the *delayed event* of τ (in this example e_1).

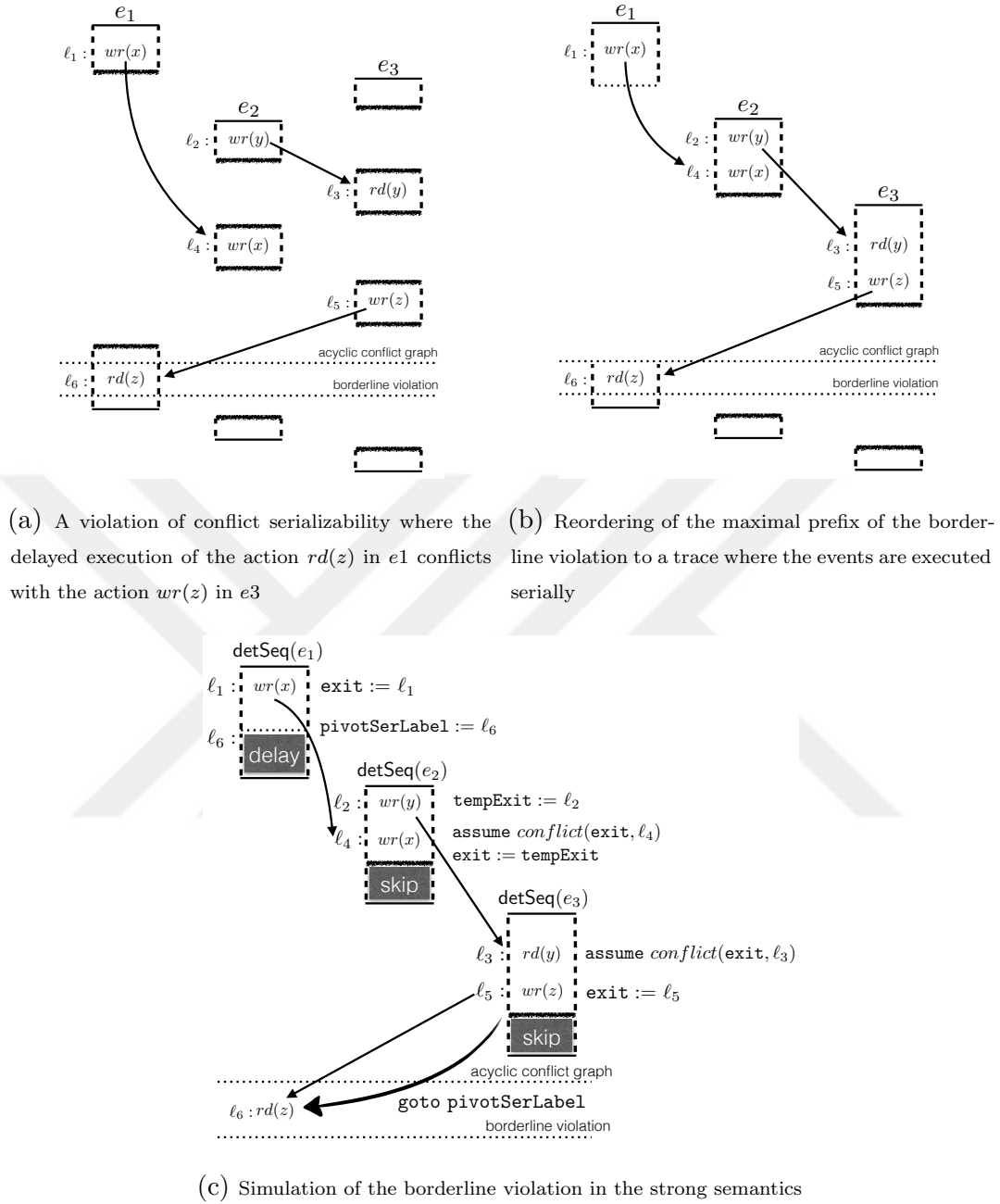


Figure 6.3: Simulating borderline conflict serializability violations with a sequential program. Boxes represent sequences of trace labels ordered from top to bottom. Actions of the same event are aligned vertically. The arrows represent all the conflicts in the trace. The grey blocks labeled by *delay*, resp., *skip*, denote sequences of actions that are delayed, resp., skipped.

6.2.1 Simulating Borderline Violations

Given a set of conflict-deterministic events E , we define a code-to-code translation from E to a set of sequential events that simulate every conflict-serializable trace and every borderline conflict serializability violation of E under the weak semantics.

As for conflict determinism, the maximal *strict* prefix of a borderline violation can be reordered to a trace where the events are executed serially, but *possibly not until completion* (since by definition, it satisfies conflict serializability). Such a reordering for the trace τ_1 is given in Figure 6.3(b). This reordering can be simulated by a sequential program that executes the conflict determinism instrumentations $\text{detSeq}(e_i)$ with $i \in [1, 3]$ instead of the original events, as shown in Figure 6.3(c). To include the pivot, the sequential program will store its label ℓ in an auxiliary variable `pivotSerLabel` when leaving e_1 and execute a `goto pivotSerLabel` after e_3 .

We use $[s]_{\text{pr}}(\mathbf{b})$ to denote a statement s that is executed at most once during the execution of the program and the boolean variable $\mathbf{b}$ is set to `true` when s gets executed. Let $\text{robSeq}^-(e)$ be the sequential event obtained from $\text{detSeq}(e)$ by:

- for a non-deterministic choice of the delayed event, the following is inserted at the beginning of each event:

```

if ( * ) then
     $[ \text{delayed} := \text{true} ]_{\text{pr}}(\text{delayedSet})$ 
    pivotSet := gotoDone := rootSet := false;

```

where `delayed` is a variable local to the event ⁴. The main boolean flags witnessing determinism violations are reset to `false` at the beginning of each event (to avoid unexpected outcomes when more than one event is executed).

- the non-deterministic choice of the pivot is implemented by the following code fragment which is added just after setting `skip` to `true` (i.e., when the non-deterministic choices $*_1$ and $*_4$ are evaluated to `true`):

⁴A variable local to an event e is a local variable of the procedure e that is transitively passed as a parameter to all the invocations made during the execution of e .

```

if ( delayed & * ) then
    [ pivotSerLabel :=  $\ell$  ]pr(pivotSerSet)

```

where **pivotSerLabel** is a global variable. This conditional checks that the pivot belongs to the delayed event.

- once the pivot has been chosen, the delayed event is not allowed to execute anymore statements and other events can non-deterministically jump to executing the pivot. This is implemented by concatenating the following to **before(s)**:

```

if ( pivotSerSet ) then
    assume ( skip | skipProc )
if ( pivotSerSet & * ) then
    [ goto pivotSerLabel ]pr(gotoSerDone)

```

Let $\text{robSeq}^-(E)$ be the set of events $\text{robSeq}^-(e)$ with $e \in E$. The following result expresses the relationship between the weak semantics of the event set E and the semantics of the set of sequential events $\text{robSeq}^-(E)$. For a trace t of $\text{robSeq}^-(E)$, $\bar{t}$ is the trace obtained from t by erasing all transition labels corresponding to statements added by the instrumentation.

Theorem 11. *For every trace τ_1 in $\llbracket E \rrbracket_w$, if τ_1 is conflict serializable or a borderline violation, then there exists a trace τ_2 in $\llbracket \text{robSeq}^-(E) \rrbracket$ such that $\tau'_1 = \bar{\tau}_2$ is a conflict-preserving permutation of τ_1 . Moreover, for every trace τ_1 in $\llbracket \text{robSeq}^-(E) \rrbracket^{\text{validMain}}$ there exists a trace τ_2 in $\llbracket E \rrbracket_w$ such that $\tau_2 = \bar{\tau}_1$.*

6.2.2 Witnessing Borderline Violations

To establish that a trace is indeed a borderline violation, the instrumentation guesses for each event a statement called *exit point* which conflicts with an action of a future event and a statement called *entry point* which conflicts with the currently recorded exit point of a previous event. The conflict is validated each time an entry point is chosen. This instrumentation is demonstrated in Figure 6.3(c). For instance, while

simulating e_1 , $wr(x)$ is guessed as the exit point and its label is recorded in the auxiliary `exit` variable. During the simulation of e_2 , $wr(x)$ is guessed as the entry point and the conflict is validated. As the simulation of e_2 shows, the exit point may occur before the entry point. In this case, the instrumentation uses an additional variable `tempExit` to store the exit point of the current event until the conflict with a previous event is validated. Once the conflict is confirmed the value of `tempExit` is copied to `exit`.

Formally, we define a new event $\text{robSeq}(e)$ that extends the instrumentation in $\text{robSeq}^-(e)$ as follows:

- to guess the exit point of an event, the following is concatenated to $\text{before}(s)$, for every statement s in a procedure $p \in \mathcal{P}(e)$:

```

if ( ! skip & delayedSet & * ) then
  if ( ! entrySet & ! delayed ) then
    [tempExit :=  $\ell(s)$ ]ev(exitSet)
  else
    [exit := tempExit :=  $\ell(s)$ ]ev(exitSet)

```

where `exit` and `tempExit` are global variables.

This guess must correspond to a statement that is not skipped and must happen after the delayed event has been chosen. If the conflict with a previous event has not been already validated (i.e., `entrySet` is `false`), and the current event is not the delayed event, then the exit point is stored in a temporary variable `tempExit`. Otherwise, it is stored directly in the variable `exit` used for confirming conflicts.

- to guess the entry point of an event and check that it indeed conflicts with the current exit point, the following is concatenated to $\text{before}(s)$, for every statement s in a procedure $p \in \mathcal{P}(e)$:

```

if ( !skip & !entrySet & !delayed & * ) then
  assume conflict(exit,  $\ell(s)$ );
  entrySet := true; interEvConf := true;

```

```

    exit := tempExit;
    if (pivotSerLabel ==  $\ell(s)$  & gotoSerDone & interEvConf) then
        assume conflict(exit, pivotSerLabel);
        error := true;

```

where the boolean expression *conflict* has been defined in Section 6.1.2. The first conditional deals with conflicts between events different from the delayed one, or conflicts *from* the delayed event to other events. The conflicts *to* the delayed event are handled by the second conditional. The boolean flag `interEvConf` is used to check that the delayed event conflicts with at least one other event in the execution.

- to ensure that entry and exit points have been correctly chosen when leaving an event, the following is added at the end of the procedure e :

```

    assume ( ( entrySet == exitSet ) | (delayed & exitSet & ! entrySet ) )

```

If an event is included in the conflict graph cycle and it is not the delayed event, then both exit and entry points must have been chosen. Otherwise, both of these guesses should not happen (i.e. `entrySet` and `exitSet` are both false). For the delayed event, only an exit point must be chosen.

Then, let $\text{robSeq}(E)$ be the set of events $\text{robSeq}(e)$ with $e \in E$. Note that the event set $\text{robSeq}(E)$ still satisfies the claim in Theorem 11.

Theorem 12. *A program P with events E satisfies conflict robustness iff $\text{robSeq}(E)$ doesn't reach a state where `error` = `true`.*

Concerning complexity, $\text{robSeq}(E)$ can be constructed in linear time and it uses a number of additional variables which is linear in the number of procedures in $\text{detSeq}(E)$. The complexity of checking conflict robustness is given by the following theorem.

Theorem 13. *Checking conflict robustness of a program P with events E , a fixed number of variables which are all boolean, and a fixed number of procedures, each*

procedure containing a fixed number of asynchronous invocations, is polynomial time decidable.

6.3 Experimental Evaluation

The goal of our experimental work [Rep,] is to show that (i) event-serializability and event-determinism violations correspond to actual bugs, and (ii) detecting these violations using the reduction to reachability in sequential programs is feasible in practice.

In this section, we present our implementation and the experimental work for:

- A dynamic event serializability checker tool that checks for the violations in the execution traces (in Section 6.3.1). Briefly, it runs the program and records the accesses of the event handlers in the execution. Then, it analyzes the collected traces offline to check for the event-serializability violations using an adaptation of the algorithm in [Alur et al., 2000, Farzan and Madhusudan, 2008]. Being a dynamic analysis tool, it does not cover all possible executions of the program running on some user input. We increase the concurrency analysis coverage by automatically running some user input scenarios for a number of times and by inserting random delays in the execution of the asynchronous tasks.
- A robustness checker tool that implements our novel method presented in Sections 6.3.2.1 and 6.3.2.2. Different from our tool in Section 6.3.1, it analyzes all possible execution traces of a program for a given sequence of user inputs. Briefly, we automatically instrument a program as presented in Chapter 6 and apply a sequential program verification tool to check for robustness. We implement the algorithm in two steps: We first check the determinism of an event handler as explained in Section 6.1. Once we verify the determinism of the event handlers, we check the event-serializability (as explained in Section 6.2) of the program for a given sequence of user inputs, i.e. event handler invocations.

We applied our tools to some applications from the FDroid (Free and Open Source software on the Android platform) [FDr,] repository, some of which are known to have concurrency bugs. In addition to detecting these concurrency bugs in the applications, our experimental work uncovered unknown concurrency bugs.

6.3.1 Checking Event-serializability of Execution Traces

We implemented a tool for Android applications by adapting the algorithm in the work [Alur et al., 2000, Farzan and Madhusudan, 2008] to event execution traces. Our tool automatically instruments an Android application and runs it on an instrumented Android runtime. The instrumented application records the execution trace which keeps the read/write accesses of the event handlers in the execution. Then, the event conflict-graph of the trace is generated and checked for cycles.

Implementation. For our event-serializability checker for Android applications, we built a modified version of Android 4.4.4 by adding: (1) an EventMapper that maintains the association between running threads and user events and (2) a TraceRecorder that logs application accesses into a trace file.

EventMapper associates each application thread with an event id whose actions are being executed in this thread.

Let *Events* be the set of event handler ids unique for each invocation of an event handler, *Threads* be the set of thread ids running in an application, *Tasks* be the set of asynchronous task invocations in an execution. EventMapper dynamically maintains the maps $ThreadToEvent : Threads \rightarrow Events$ and $TaskToEvent : Tasks \rightarrow Events$ to associate each thread and asynchronous invocation to an event id.

The association of the threads and tasks to event identifiers are updated each time (i) an event handler is triggered and dispatched on the main thread: (ii) an asynchronous task is spawned (e.g. an `AsyncTask` is created, a `Runnable` task or a `Message` to be handled is posted to a thread's task buffer, a `Runnable` is invoked on a fresh background thread) and (iii) a thread starts executing an asynchronous task (e.g. a background thread starts executing `doInBackground` method of an `AsyncTask`,

the main thread starts executing asynchronously invoked `onPostExecute` method, or a fresh thread starts running a `Runnable` task.)

The maps *ThreadToEvent* and *TaskToEvent* are updated as follows:

- (i) An event handler is dispatched on the main thread (with thread id $Thread_{main}$):

$Events := Events \cup \{eid\}$ where eid is a fresh event id

$ThreadToEvent[Thread_{main}] := eid.$

- (ii) When an asynchronous task is created during the execution of the thread $Thread_i$, the event id associated with the currently executing thread $Thread_i$ is assigned to the spawned asynchronous task.

$Tasks := Tasks \cup \{Task_i\}$ where $Task_i$ is a fresh id for the spawned task,

$TaskToEvent[Task_i] := ThreadToEvent[Thread_i]$

- (iii) When a thread $Thread_i$ starts executing an asynchronous task $Task_i$, the event id associated with $Task_i$ is assigned to $Thread_i$ since it executes the asynchronous task transitively spawned by this event id.

$ThreadToEvent[Thread_i] := TaskToEvent[Task_i]$

Different threads in an application may be assigned to different event handlers ids at an instant of time of the execution. For instance, while the main thread is processing the handler of the second user input and hence currently associated with the id of the second event, a background thread may be running an asynchronous task invoked by the first input and be associated with the id of the first event.

A thread may be assigned to different event handler ids during the execution of a program. Consider an event handler invocation with id eid is running on the main thread, i.e. $ThreadToEvent[Thread_{main}] = eid$. Assume that this event handler creates an `AsyncTask` with id $Task_i$, then the created `AsyncTask` object is associated with this event id, i.e. $TaskToEvent[Task_i] := eid$. When a background thread

executes the `doInBackground` task of this `AsyncTask`, the background thread $Thread_i$ is associated to its event id, i.e. $ThreadToEvent[Thread_i] := TaskToEvent[Task_i]$. Similarly, the main thread gets associated to eid when it executes the `onPostExecute` of this `AsyncTask` object. Notice that the main thread may run some other tasks with different event ids in between executing the event handler invocation with the id eid and its asynchronous task `onPostExecute`.

TraceRecorder records the object field accesses in an application, together with the corresponding event ids of the event handlers that give rise to that access. It keeps a list of accesses and writes them into a file.

An application under test is automatically instrumented using Soot framework [Arzt et al., 2013] by inserting calls to methods of *EventManager* and *TraceRecorder*. With the goal of different runs of the same user interaction with the application resulting in different access interleavings, we apply a simple “noise making” approach. We instrument an application to insert random delays into the asynchronous tasks which run in the background threads. After instrumentation, these tasks either run without waiting or they wait for some amount of time before they start their execution.

To check for the event-serializability violations in an execution trace, we build the conflict-event graph of the trace and check whether it is acyclic. For cycle detection, we implement the algorithm presented in [Farzan and Madhusudan, 2008].

Benchmarks and tests. For each application under analysis, we specified a number of user scenarios and wrote them as *monkeyrunner* test scripts. We automatically ran each user input scenario 15-100 times to exercise different task interleavings, exploring 1375 traces in total. The experimental results for the serializability tests are shown in Table 6.1.

True bugs. Four of the benchmarks had traces with conflict-event serializability violations (marked with a “Y” in Table 6.1) which we concluded were true bugs (and true event-serializability violations) after examining the application code and the consequences of the conflict event-serializability violation. Briefly, *v1111e* throws a concurrent modification exception, in *irccloud* the application sends wrong message

contents, in `bookworm`, the `searchPosition` variable saved in the application cache is stale, and in `diode`, the `mAfter` string on the UI thread, indicating which reddit threads come after the ones currently loaded, shows stale information. These applications are not life critical but illustrate that, in addition to exceptions, serializability violations can cause corruption of persisted data or data sent onto the network.

Avoidable false alarms. In three of the applications, we determined the conflict-serializability violations *not* to be bugs and *not* to be actual serializability violations. (marked with a “N”). In `apollo`, concurrent tasks read from and wrote to certain `mImageCache` fields. After initialization, the same value is written to this field, in a redundant manner. All conflict-serializability violations in `apollo` were due to this and *not* serializability violations. A simple modification to our checker which ignores overwriting a variable with the same value would avoid this false positive. In `comicsreader`, the violation occurs because the `onPause` event handler resets an `mInit` field redundantly. As we confirmed by inspection and experimentally, in all possible executions, after being updated by `onPause`, `mInit` is always overwritten before being read anywhere. Removing this redundant write also eliminates the violations. In `grtgtsfs`, the executions with conflict-serializability violations are, in fact, serializable and are not buggy. When the user triggers a search quickly after another one, the two instances of the event handler create two async tasks. Both tasks may end up performing the same, later search. While technically this is not a serializability violation, the application performs redundant work and the code could be improved. In all three cases, we believe it is worthwhile to report the conflict-serializability violations to the programmer, because fixing them would lead to improved code.

modCount. For `vlille`, conflict-serializability violations due to the `modCount` variable of a `List` object correctly signalled executions as buggy. In `vibevault`, `grtgtsfs`, and `bart`, there were small numbers of executions that experienced violations due to `modCount` but did *not* result in any exceptions. Although we have not been able to determine the root cause of these violations, we believe accessing the `List` data structure in multiple threads is potentially problematic.

App		#tr	size	#E	#obj	#v	t(s)	B
com.andrew.apollo	1	100	799	26	203	83	0.02	N
	2	100	685	20	182	80	0.02	N
	3	100	722	24	189	84	0.02	N
com.vlille.checker	1	50	903	14	222	49	0.02	Y
	2	50	3035	18	567	0	0.06	
pro.dbro.bart	1	50	3324	17	683	1	0.07	-
	2	50	3692	15	667	0	0.08	
net.kw.shrdlu.grtgfts	1	60	1408	15	143	2	0.02	-
	2	50	1221	15	134	1	0.02	-
	3	50	832	11	101	0	0.01	
	4	50	696	10	91	17	0.01	N
com.android.ircloud	1	30	4595	16	1023	30	0.10	Y
	2	30	4981	32	982	23	0.28	-
com.code.android.vibevault	1	25	9392	12	1553	1	0.86	U
	2	25	39920	14	7265	1	2.38	U
	3	25	27255	5	4496	0	0.88	
net.kervala.comicsreader	1	100	831	57	158	11	0.02	N
	2	100	411	35	107	0	0.01	
	3	100	582	30	157	3	0.01	-
com.totsp.bookworm	1	50	8237	27	1075	0	0.15	
	2	50	5682	20	654	8	0.10	Y
	3	50	3763	19	964	0	0.60	
	4	50	4520	25	1185	0	0.08	
in.shick.diode	1	15	3185	4	629	0	0.47	
	2	15	7241	4	1784	15	1.12	Y

Table 6.1: Experimental data for checking conflict-event serializability. The columns represent the total number of collected traces, the avg trace length, the avg number of events, the avg number of objects, the number of violations in the considered trace set, and the avg time (in seconds) required for checking serializability of one trace. Time measurements are made on a MacBookPro 2.5 GHz Intel Core i7 Quad core machine.

6.3.2 Checking Robustness Against Asynchrony

We implemented our conflict-robustness checking algorithm as an instrumentation to Java programs. We use the Soot framework [Arzt et al., 2013] to implement the instrumentation required for robustness checking. The reachability of the error state in the instrumented sequential program is verified using Java Path Finder (JPF) [JPF,].

The application code for reflection, dependency to external libraries (e.g., for http connection, gson object processing, analytics tracker, maps), and the code which only effects the display (e.g., displaying web pages, animation, custom graphics) is

Application	Event handler	#inst	#c	#m	#r/w	#(*)	t(m:s)	Det?
aarddict.android	create activity	1307780	177	3016	90	1428	0:01	Y
	lookup word	77203	222	3604	60	103	< 1s	Y
	scan sd	21334	167	2941	15	21	< 1s	Y
ca.mimic.apphangar	select item	58908	222	3560	48	70	< 1s	Y
	update icon pack	13308927	264	4004	95	28833	00:33	N
com.totsp.bookworm	generate cover	34528	194	2928	30	41	< 1s	Y
	retrieve cover	36789	213	3440	31	41	< 1s	Y
	save edits	63017	189	3015	108	158	< 1s	Y
	search book	53250	185	3012	50	69	< 1s	Y
net.kw.shrdlu.grtgfts	fav stops	53995	162	2885	142	113	< 1s	Y
	process bustimes	65945	159	2749	105	168	< 1s	Y
	search route	55077	167	2968	34	67	< 1s	Y
	search stop	56742	168	2968	52	75	0:01	Y
com.android.ircloud	save prefs	103344	293	3478	18	15	< 1s	Y
	save settings	102868	293	3478	17	13	< 1s	Y
	select buffer	136224103	379	4330	761	260605	8:04	Y
	send message	162682	356	4140	171	77	< 1s	Y
com.ville.checker	load stations	971665	404	5808	236	131	0:01	Y
	load favorites	9583	141	2400	37	0	< 1s	Y
	update stations	975974	416	5905	265	131	0:01	Y

Table 6.2: Experimental data for conflict determinism. The last column lists whether the event is found to be conflict deterministic.

eliminated. The remaining code factors out the variables that does not effect the concurrent behavior of the program and keeps the program logic.

We define an event as a procedure which is invoked by the Android app in order to initialize an activity, in response to an user input (e.g., clicking on a button, writing text, navigating back) or a system input (e.g., location change, network disconnect). Our tool receives as input a driver class which initiates the application (possibly some initial state data) and invokes a set of events. The tool checks conflict-robustness for the set of executions defined by the driver class. In our experiments, we also consider the causality constraints between events, e.g., the event handler of a UI component can not be invoked if it is not visible on the screen.

6.3.2.1 Event-determinism experiments

Table 6.2 lists the experimental data related to the conflict-deterministic checking algorithm. To give an idea about the size of the events used in this experiment, we list the number of analyzed instructions (`#inst`), loaded classes (`#c`) and methods (`#m`). We also give the number of read/write accesses processed by the instrumentation (`#r/w`) and the number of resolved non-deterministic choices (`#(*)`).

We have applied this algorithm to various event-handlers and all but one are found to be deterministic. The `iconPackUpdated` event invokes a few asynchronous procedures, two of them being involved in a determinism violation as presented in Section 5.2. The pivot of the violation is a write access to the `mAppRowAdapter` variable by an asynchronous procedure running in the background, and the root is a read on the same variable made by an asynchronous procedure running on the main thread.

6.3.2.2 Event-serializability experiments

Table 6.3 shows experimental data for conflict-serializability checking.

True bugs. Four of the benchmarks had traces with conflict serializability violations which we concluded were true bugs (and true event-serializability violations) after examining the code and the consequences of these violations.

The event serializability violation in `aarddict` app occurs between the initialization of the activity (initializes the UI components and starts the dictionary service to load the dictionaries) and an event handler to lookup a word. The lookup cannot retrieve the requested word if the service gets initialized after the lookup. In this benchmark, the pivot of the serializability violation is a write access to a variable `dictionaryService` in an asynchronous procedure invoked on the main thread that conflicts with the asynchronous procedure invoked on a background thread by the second event handler.

We detected an event serializability violation in the `bookworm` app between the events dealing with user inputs to search for a book and navigating back to the previous screen. In this violation, while the first event handler performs the search in

Application	Seq	#inst	#c	#m	#r/w	#(*)	t(m:s)	Ser?	Bug?
aarddict.android	1	1084371993	224	3620	154	1764359	23:12	N	Y
	2	101776570	169	2957	100	195370	1:42	N	Y
com.totsp.bookworm	1	22701600	183	2801	202	77614	0:42	Y	-
	2	19179949	183	2801	201	61896	0:33	Y	-
	3	1094300968	189	3016	286	3494089	33:51	Y	-
	4	3547795	188	3029	131	15961	0:08	N	Y
net.kw.shrdlu.grtgfts	1	74082801	168	2969	123	279857	2:04	Y	-
	2	-	-	-	149	-	> 1h	-	-
	3	1130239	139	2692	77	4712	0:02	Y	-
	4	60736622	170	2984	161	163236	1:21	N	N
com.android.ircccloud	1	33713083	293	3479	141	147000	2:55	Y	-
	2	1761539	293	3479	140	7851	0:10	Y	-
	3	171715464	294	3485	147	534338	08:51	Y	-
	4	-	-	-	-	2110	> 1h	-	-
	5	-	-	-	-	902	> 1h	-	-
	6	54556857	358	4165	849	208076	5:28	N	Y
	7	11104756	357	4154	833	39599	0:59	N	Y
com.vlille.checker	1	48935337	406	5824	286	143461	3:05	N	Y
	2	394535226	406	5824	292	1319041	28:52	N	N

Table 6.3: Experimental data for checking conflict serializability. The last two columns list whether the benchmark is conflict serializable and whether a detected violation is a true bug or not.

the background and not yet updated the `currSearchTerm` variable, the second event handler saves the stale `currSearchTerm` variable in the cache. The pivot of the violation is a write access to the current search term in an asynchronous procedure invoked on the background thread.

A serializability violation detected in the `ircccloud` app is presented in Section 5.2, which causes the app to send wrong message content. The pivot is a read access to the message text in an asynchronous procedure invoked on a background thread that conflicts with a write access in the double click event. A similar violation occurs in another user input sequence in which the user types some text after pressing the “send” key.

In the `vlille` benchmark, the detected event serializability violation in the first line is due to the asynchrony bug presented in Figure 4.2 in Chapter 4. It occurs

when the user removes an item from his favorites list while the items are being loaded. The application throws a concurrent modification exception when the removal in the second event handler interleaves with the asynchronous procedure in the background thread. This violation takes more than one hour to be caught and it is checked with a hint (the object field name of the pivot access) to detect the violation in shorter time.

Avoidable false alarms. In the `grttfts` benchmark, we determined the conflict-serializability violation *not* to be a bug and *not* to be an actual serializability violation. (Conflict-serializability is not a necessary condition for serializability.) This violation is triggered by two different users making two queries one after another. In an execution where the second event handler overwrites the query before the first event handler reads it in the background, both asynchronous procedures end up performing the same, later search.

Inter-related events. Some event handlers intervene the execution of another event by design. For such inter-related events, the event-serializability violation might not be a bug. The `vlille` benchmark has an example for this case (the second row on the table). In this scenario, the user navigates back while the application is loading a list of items asynchronously in a background thread. The event handler for back navigation sets the `mCancelled` flag of the `AsyncTask`. If this flag is set, the first event handler does not invoke the `AsyncTask`'s asynchronous `onPostExecute` procedure.

6.4 Discussion of Special Cases

In this section, we discuss how our correctness criterion robustness against asynchrony tackles some special cases in event-driven asynchronous programs.

First, we consider event handlers which wait for the completion of some tasks. Although blocking wait is not a common programming practice inside event-driven applications, some programs can have it as it is allowed by the programming languages. It is important to specify the behavior of a robust program for the case with task waiting since our standard definition of strong semantics lead to blocking program execution in certain cases. We identify these cases and specify a slight extension of

the strong semantics to provide progress of the programs.

Secondly, we discuss a special case of event handlers, which intervene the execution of some other event handlers by design. Recall that robustness against asynchrony requires the execution event handlers to be equivalent to an execution in which the event handlers run serially without interference from each other. The special case of inter-related events are intended to interfere with each other and hence violate robustness.

6.4.1 *Waiting for an Asynchronous Task*

A common pattern to provide responsiveness of event driven programs is to avoid blocking by waiting for the completion of a task but to invoke a task asynchronously to be executed after the completion of that task. For example, `AsyncTask` class in Android encourages this style of programming and makes it easy to write such programs by abstracting the details of asynchronous task invocation and multithreading. The programmer only implements `doInBackground` to run a long running job in a background thread asynchronously, `onPreExecute` to run on the main thread (e.g. to prepare some application variables before the background task), `onProgressUpdate` that runs asynchronously on the main thread to update the UI with the current progress and `onPostExecute` to run asynchronously on the main thread (e.g. to display the result on the UI or update the application variables) after the completion of the background. Although blocking by waiting for a task is not encouraged, it is allowed in the programming language and some event-driven asynchronous programs can involve task waiting.

As robustness against asynchrony of a program requires equivalence to its execution under the strong semantics, we discuss the characteristics of the execution of a program with task waiting under the strong semantics. We distinguish two different cases where an event handler can wait for the completion of another task:

- *Waiting for an asynchronous task created in a previous event:* Our definition of strong semantics entails a valid and an intuitive execution of such programs by

definition. It guarantees the completion of all asynchronous tasks invoked in an event handler before dispatching another event handler. By definition, at the time the waiting task is executing, the asynchronous task of a previous event handler has already completed. Robustness against asynchrony requires the behavior of a program to be equivalent to this execution in the strong semantics.

- *Waiting for an asynchronous task invoked in the same event:* Recall that the strong semantics executes an event handler procedure to completion before dispatching its asynchronous invocations. To guarantee the progress of an event handler which waits for the completion of its asynchronous invocations, we extend the strong semantics by slightly modifying the behavior of the task buffer. When a task waiting statement is reached in the execution of a task, the execution of the current procedure freezes and all the tasks in the buffer are executed in the order of their invocations. Then, the frozen procedure is resumed to execute (the frozen procedure is guaranteed to proceed at this point of execution since all asynchronous tasks in the buffer that are possibly waited for are completed).

6.4.2 Inter-related Events

Some event handlers are supposed to interfere the execution of another event. A typical example user input for such an event handler is to cancel some operation (e.g. a download operation) spawned by another event handler. Similarly, navigating back to a previous screen may cause the abortion of the asynchronous tasks invoked in a previous event handler and not needed to be executed in the back-navigated screen. The intended outcome of the execution of such event handlers may not be serializable, hence a detected robustness violation may not cause a buggy execution. For such inter-related events, the intended execution of the event handlers may signal an event-serializability violation that does not cause a buggy execution. Our experimental work in Section 6.3.2.2 presents such an event-serializability violation where an event handler for navigating back cancels the asynchronous task of a previous event.

Currently, our notion considers program robustness without the inter-related events.

As a future work, it can be modified to consider such events with task abortion.

6.5 Related Work

Several static analysis tools target the detection of problems related to security and information flow in Android applications [Li et al., 2015, Gordon et al., 2015, Arzt et al., 2014]. Dynamic techniques [Anand et al., 2012b, Machiry et al., 2013b, Azim and Neamtiu, 2013b, Mirzaei et al., 2012, Liu and Xu, 2013, vander Merwe et al., 2012], including those that employ symbolic execution, have been developed to generate user interface events in order to cover different features in the application code. These tools do not explicitly target concurrency errors and sometimes suffer due to concurrency-related side effects.

Most existing tools for detecting concurrency errors in Android apps investigate race detection [Maiya et al., 2014, Hsiao et al., 2014, Bielik et al., 2015]. Section 6.5.1 provides a brief comparison of robustness against asynchrony and data race freedom violations.

Recent work [Safi et al., 2015] proposes a static analysis to detect “anomalies” in event driven programs, i.e. accesses to the same memory location by more than one event handlers. Since many events access shared memory locations, this approach produces many false alarms, but programs without anomalies are conflict-event serializable. The works in [Lin et al., 2014, Lin et al., 2015] refactor applications by moving long running jobs to asynchronous tasks and transform improperly-used asynchrony constructs into correct constructs. Ensuring transformed asynchronous tasks do not race with their callers lends support to our work as it guarantees event-determinism.

The works in [Emmi et al., 2012b, Emmi et al., 2011c, Emmi et al., 2014, Ozkan et al., 2015] target exploring interesting subsets of executions and schedules for asynchronous programs, that offer a large coverage of the execution space. This is orthogonal to the focus of our work which is to investigate correctness criteria for event-driven asynchronous programs.

Conflict serializability [Papadimitriou, 1979] has been introduced in the context of database transactions and since then used as a tractable approximation of atomicity.

We use serializability to formalize the fact that event handlers behave as if they were executed in isolation, without interference from others. While in other uses of serializability the transactions are sequential, in our case a single invocation of an event handler consists of several asynchronous procedures that can interleave arbitrarily in between them. [Farzan and Madhusudan, 2008, Farzan and Madhusudan, 2009] and [Bouajjani et al., 2013] investigate decision procedures for conflict serializability of finite-state concurrent models while checking serializability in general has been approached using both static, e.g., [Flanagan et al., 2008a, Hatcliff et al., 2004, von Praun and Gross, 2004, Yi et al., 2012], and dynamic tools, e.g., [Flanagan and Freund, 2008, Wang and Stoller, 2006, Flanagan et al., 2008b, Sinha et al., 2011].

Determinism has been largely advocated in the context of concurrent programs, e.g., [Bocchino et al., 2009, Steele, 1990], since it simplifies the debugging and verification process. Prior work has introduced static verification techniques, e.g., [Jr. et al., 2009] but also dynamic analyses based on testing, e.g., [Burnim and Sen, 2010, Sadowski et al., 2009]. Differently from prior work, we provide a methodology for checking determinism of event-driven asynchronous programs that ultimately reduces to a reachability problem in a sequential program.

6.5.1 Robustness Against Asynchrony vs Data Race Freedom Violations

Race conditions are low-level symptoms for a much broader class of concurrent programs which are often not indicative of actual programming errors. In this work, we characterize and detect higher-level concurrency errors in Android programs. Robustness violations are incomparable with data-race freedom violations. Data races do not generally imply cyclic data dependencies among events, and cyclic data dependencies do not imply data races: e.g., surrounding each individual memory access within a cycle by a common lock eliminates possible races, but preserves cycles. Furthermore, checking conflict robustness is fundamentally more efficient than checking for data race freedom. Conflict event serializability requires tracking events, while data race freedom requires tracking individual program actions like reads and writes,

which greatly outnumber events. Moreover, conflict robustness reduces to reachability in *sequential* programs, yielding significantly lower asymptotic complexity.

In this section, we show that robustness against asynchrony more accurately models the high-level symptoms of concurrency errors in event-driven asynchronous programs. Data race freedom models low-level symptoms for a much broader class of concurrent programs which are often not indicative of actual programming errors. We show that various types of false positives detected in the data race checking do not cause a robustness violation.

6.5.1.1 False Positives in Data Race Checking

We classify the false positives in data race checking into two categories: (i) The detected races that are not actually data races (ii) The data races that do not cause a bug.

The false positives that are not indeed races:

- Data race detection relies on the happens-before relation for a language or a platform. The missing orderings in the definition of the happens before relation for Android platform results in detecting false races.
- Some false races might arise due to the implementation of race checkers (e.g dereferencing to an incorrect pointer).

The false positives that are races but not bugs:

- The commutative operations (e.g. on adding an existing element and getting it from a list) might produce data races but do not cause buggy executions. Such executions are captured as robust (though not conflict robust) against asynchrony.
- The work in [Hsiao et al., 2014] identifies and filters some data races that do not cause buggy executions. For example, an event handler's access to check whether a pointer is null before using it races with another event handler's concurrent

access that sets the pointer. Such a program does not violate robustness against asynchrony if there are no cyclic dependencies between the event handlers.

6.5.1.2 True Bugs Detected by Data Race Checking

We conclude that the following bugs detected and presented by race checking tools are also robustness violations. We listed only the examples whose source code is available so that we can validate the robustness violation by examining the object accesses causing the violation.

- The racy execution in `aarddict` app presented in [Maiya et al., 2014] is detected as an event serializability violation in our experimental work. It violates the serializability of events (1) initialization of the app that loads the dictionary in the background and (2) looking up a word that accesses empty dictionaries.
- The racy execution in `OI manager` presented in [Bielik et al., 2015] is caused by the write accesses to a view element. The problematic execution has a violation to the event serializability between the user inputs to display the list of items and scroll to display more.
- A harmful race in a JavaScript program is presented in [Raychev et al., 2013] which violates the serializability of events (1) initialization of the web page and (2) clicking on a button.

6.5.1.3 True Bugs that are allowed by the strong semantics

Data race checkers signal a violation, when there is a race between two events that can be invoked in a non-deterministic order. If any ordering of these events are allowed in the strong asynchronous semantics, the execution of these two events do not cause a robustness violation.

6.5.1.4 True Bugs Cannot Be Detected by Data Race Checking

Some robustness violations are data race free. An execution of two event handlers with cyclic dependencies are not robust against asynchrony but it is race free if the memory accesses are protected by locks.



Chapter 7

CONCLUSION

In this dissertation, we characterized the correctness of event-driven asynchronous programs as *robustness against asynchrony*. We presented a novel, efficient algorithmic approach for checking robustness based on polynomial time reductions to reachability problems in sequential programs. We implemented our robustness checking algorithm for Android applications and detected some known and unknown concurrency bugs in several open source applications.

We conclude the thesis by summarizing the features of robustness against asynchrony correctness criterion:

- Robustness is independent of the application logic or the execution platform. Intuitively, a program is asynchronously robust, if it exhibits the same set of behaviors regardless of the asynchrony and interleaving shared memory accesses of the concurrent tasks in the execution.
- Robustness against asynchrony provides a high-level correctness criterion. Different from the symptomatic concurrency bug detection techniques such as data race detection, it does not produce high number of false positives. For instance, a bug free execution of racy memory accesses of two event handlers (e.g. a write access to assign an object to a null pointer and a concurrent read access to check whether the pointer is null before using it) does not cause robustness violation if there are no cyclic dependencies between the two event handlers.
- The real concurrency bugs in the open source applications can be captured as robustness violations. Moreover, unknown concurrency bugs can be uncovered by

robustness checking, which shows the validity of robustness against asynchrony as a correctness criterion.

- Robustness can be formulated by the conjunction of two intuitive properties: event-serializability and event-determinism. Event serializability requires the executions of a sequence of event handlers to be equivalent to a serial execution of them in which an event handler is dispatched only after the completion of all asynchronous tasks of the previous event. Event determinism focuses on a single event handler and requires it to produce a deterministic outcome regardless of the interleaving execution of its asynchronous invocations.
- Robustness can be algorithmically and efficiently checked by means of its decomposition into event serializability and event determinism. The conflict robustness of event-driven asynchronous programs with high concurrency with an unbounded number of asynchronous tasks can be reduced to the sequential program analysis. This reduction allows to leverage existing sequential program analysis tools to check conflict-robustness of event-driven asynchronous programs.

In our experimental work on open source Android applications, we show that robustness against asynchrony can be effectively used for the detection of concurrency bugs in the event-driven asynchronous programs.

Limitation and Future Work. Robustness checking may raise false positives in the analysis of inter-related event handlers, which are intended to interfere the execution of another event. Robustness may be violated in a correct execution of such event handlers, since the execution is not event serializable by design. A possible direction for the future work may be to extend the characterization and the verification of the robustness to prevent false positives in the analysis of the inter-related events.

BIBLIOGRAPHY

- [FDr,] F-Droid - Free and Open Source App Repository. <http://f-droid.org/>. Accessed: 2015-09-07.
- [Irc, a] <http://github.com/irccloud/android/tree/9e2f5cf04e>.
- [Vli,] <http://github.com/ojacquemart/vlillechecker/issues/60>.
- [Rep,] <https://github.com/async-robustness/robustness-checker>.
- [App,] <https://github.com/corcoran/Hangar/blob/master/Apphangar/src/main/java/ca/mimic/apphangar/Settings.java#l857>.
- [Irc, b] <https://github.com/irccloud/android/commit/c81f3374>.
- [Msd,] <http://social.msdn.microsoft.com/forums/windowsapps/en-us/292731da-a979-4519-9a85-d804f8987f61/async-hell-intermittent-loading-of-collection-before-binding>.
- [Cod,] <http://www.codeproject.com/articles/562021/asynchronous-models-and-patterns>.
- [JPF,] Java pathfinder. <http://babelfish.arc.nasa.gov/trac/jpf/>. Accessed: 2015-09-07.
- [Alur et al., 2000] Alur, R., McMillan, K. L., and Peled, D. (2000). Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188.
- [Anand et al., 2012a] Anand, S., Naik, M., Harrold, M. J., and Yang, H. (2012a). Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT*

- 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA. ACM.
- [Anand et al., 2012b] Anand, S., Naik, M., Harrold, M. J., and Yang, H. (2012b). Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA. ACM.
- [Arzt et al., 2013] Arzt, S., Rasthofer, S., and Bodden, E. (2013). Instrumenting android and java applications as easy as abc. In *Runtime Verification*, pages 364–381. Springer.
- [Arzt et al., 2014] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). 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. ACM.
- [Atig et al., 2012] Atig, M. F., Bouajjani, A., Emmi, M., and Lal, A. (2012). Detecting fair non-termination in multithreaded programs. In *CAV '12: Proc. 24th International Conference on Computer Aided Verification*, LNCS. Springer.
- [Atig et al., 2009] Atig, M. F., Bouajjani, A., and Qadeer, S. (2009). Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 107–123.
- [Azim and Neamtiu, 2013a] Azim, T. and Neamtiu, I. (2013a). Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems*

- Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA. ACM.
- [Azim and Neamtiu, 2013b] Azim, T. and Neamtiu, I. (2013b). Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA. ACM.
- [Barnett et al.,] Barnett, M., Leino, K. R. M., Moskal, M., and Schulte, W. Boogie: An intermediate verification language. <http://research.microsoft.com/en-us/projects/boogie/>.
- [Bielik et al., 2015] Bielik, P., Raychev, V., and Vechev, M. (2015). Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA. ACM.
- [Bocchino et al., 2009] Bocchino, Jr., R. L., Adve, V. S., Adve, S. V., and Snir, M. (2009). Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA. USENIX Association.
- [Bouajjani and Emmi, 2012a] Bouajjani, A. and Emmi, M. (2012a). Analysis of recursively parallel programs. In *POPL '12: Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–214. ACM.
- [Bouajjani and Emmi, 2012b] Bouajjani, A. and Emmi, M. (2012b). Bounded phase analysis of message-passing programs. In *TACAS '12: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer.

- [Bouajjani and Emmi, 2014] Bouajjani, A. and Emmi, M. (2014). Bounded phase analysis of message-passing programs. *International Journal on Software Tools for Technology Transfer*, 16(2):127–146.
- [Bouajjani et al., 2013] Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2013). Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer.
- [Bouajjani et al., 2011] Bouajjani, A., Emmi, M., and Parlato, G. (2011). On sequentializing concurrent programs. In *SAS '11: Proc. 18th International Symposium on Static Analysis*, volume 6887 of *LNCS*, pages 129–145. Springer.
- [Burnim and Sen, 2010] Burnim, J. and Sen, K. (2010). Asserting and checking determinism for multithreaded programs. *Commun. ACM*, 53(6):97–105.
- [Chen et al., 2007] Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and min Wang, Y. (2007). A systematic approach to uncover security flaws in gui logic. In *In IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society.
- [Choi et al., 2013] Choi, W., Necula, G., and Sen, K. (2013). Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '13*, pages 623–640, New York, NY, USA. ACM.
- [Desai et al., 2015] Desai, A., Qadeer, S., and Seshia, S. A. (2015). Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 73–83, New York, NY, USA. ACM.
- [Elmas et al., 2007] Elmas, T., Qadeer, S., and Tasiran, S. (2007). Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN*

- Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, New York, NY, USA. ACM.
- [Emmi and Lal, 2012] Emmi, M. and Lal, A. (2012). Finding non-terminating executions in distributed asynchronous programs. In *SAS '12: Proc. 19th International Static Analysis Symposium*, LNCS. Springer.
- [Emmi et al., 2012a] Emmi, M., Lal, A., and Qadeer, S. (2012a). Asynchronous programs with prioritized task-buffers. In *SIGSOFT FSE '12: Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, page 48. ACM.
- [Emmi et al., 2012b] Emmi, M., Lal, A., and Qadeer, S. (2012b). Asynchronous programs with prioritized task-buffers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 48:1–48:11, New York, NY, USA. ACM.
- [Emmi et al., 2014] Emmi, M., Ozkan, B. K., and Tasiran, S. (2014). Exploiting synchronization in the analysis of shared-memory asynchronous programs. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 20–29, New York, NY, USA. ACM.
- [Emmi et al., 2011a] Emmi, M., Qadeer, S., and Rakamaric, Z. (2011a). Delay-bounded scheduling. In *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM.
- [Emmi et al., 2011b] Emmi, M., Qadeer, S., and Rakamarić, Z. (2011b). Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422, New York, NY, USA. ACM.
- [Emmi et al., 2011c] Emmi, M., Qadeer, S., and Rakamarić, Z. (2011c). Delay-bounded scheduling. *SIGPLAN Not.*, 46(1):411–422.

- [Farzan and Madhusudan, 2008] Farzan, A. and Madhusudan, P. (2008). Monitoring atomicity in concurrent programs. In *CAV '08: Proc. 20th Intl. Conf. on Computer Aided Verification*, volume 5123 of *LNCS*, pages 52–65. Springer.
- [Farzan and Madhusudan, 2009] Farzan, A. and Madhusudan, P. (2009). The complexity of predicting atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 155–169.
- [Flanagan and Freund, 2008] Flanagan, C. and Freund, S. N. (2008). Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109.
- [Flanagan and Freund, 2009] Flanagan, C. and Freund, S. N. (2009). Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 121–133, New York, NY, USA. ACM.
- [Flanagan et al., 2008a] Flanagan, C., Freund, S. N., Lifshin, M., and Qadeer, S. (2008a). Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4).
- [Flanagan et al., 2008b] Flanagan, C., Freund, S. N., and Yi, J. (2008b). Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 293–303.
- [Ganty and Majumdar, 2012] Ganty, P. and Majumdar, R. (2012). Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6.

- [Garg and Madhusudan, 2011] Garg, P. and Madhusudan, P. (2011). Compositionality entails sequentializability. In *TACAS '11: Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*, pages 26–40. Springer.
- [Gomez et al., 2013] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 72–81, Piscataway, NJ, USA. IEEE Press.
- [Gordon et al., 2015] Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. (2015). Information-flow analysis of android applications in droidsafe. In *2015 Network and Distributed System Security (NDSS) Symposium*.
- [Hatcliff et al., 2004] Hatcliff, J., Robby, and Dwyer, M. B. (2004). Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 175–190.
- [Hsiao et al., 2014] Hsiao, C.-H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C. L., Pokam, G. A., Chen, P. M., and Flinn, J. (2014). Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 326–336, New York, NY, USA. ACM.
- [Hu and Neamtiu, 2011] Hu, C. and Neamtiu, I. (2011). Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST '11*, pages 77–83, New York, NY, USA. ACM.
- [Jhala and Majumdar, 2007] Jhala, R. and Majumdar, R. (2007). Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGPLAN-*

- SIGACT Symposium on Principles of Programming Languages*, pages 339–350. ACM.
- [Jhala and Majumdar, 2009] Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54.
- [Jr. et al., 2009] Jr., R. L. B., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. (2009). A type and effect system for deterministic parallel java. In Arora, S. and Leavens, G. T., editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 97–116. ACM.
- [Kidd et al., 2010] Kidd, N., Jagannathan, S., and Vitek, J. (2010). One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN '10: Proc. 17th International Workshop on Model Checking Software*, volume 6349 of *LNCS*, pages 245–261. Springer.
- [Kozen, 1977] Kozen, D. (1977). Lower bounds for natural proof systems. In *FOCS '77: Proc. 18th Annual Symposium on Foundations of Computer Science*, pages 244–266. IEEE Computer Society.
- [La Torre et al., 2009] La Torre, S., Madhusudan, P., and Parlato, G. (2009). Reducing context-bounded concurrent reachability to sequential reachability. In *CAV '09: Proc. 21st International Conference on Computer Aided Verification*, volume 5643 of *LNCS*, pages 477–492. Springer.
- [La Torre et al., 2010] La Torre, S., Madhusudan, P., and Parlato, G. (2010). Model-checking parameterized concurrent programs using linear interfaces. In *CAV '10: Proc. 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 629–644. Springer.

- [Lal et al., 2012] Lal, A., Qadeer, S., and Lahiri, S. K. (2012). A solver for reachability modulo theories. In *Proceedings of the 24th International Conference on Computer Aided Verification, CAV'12*, pages 427–443, Berlin, Heidelberg. Springer-Verlag.
- [Lal and Reps, 2009] Lal, A. and Reps, T. W. (2009). Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97.
- [Li et al., 2015] Li, L., Bartel, A., Bissyande, T. F., Klein, J., Traon, Y. L., Arzt, S., Rasthofer, S., Bodden, E., Outeau, D., and McDaniel, P. (2015). Iccta: Detecting inter-component privacy leaks in android apps. In *2015 International Conference on Software Engineering (ICSE)*.
- [Lin et al., 2015] Lin, Y., Okur, S., and Dig, D. (2015). Study and refactoring of android asynchronous programming. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*.
- [Lin et al., 2014] Lin, Y., Radoi, C., and Dig, D. (2014). Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 341–352, New York, NY, USA. ACM.
- [Liu and Xu, 2013] Liu, Y. and Xu, C. (2013). Veridroid: Automating android application verification. In *Proceedings of the 2013 Middleware Doctoral Symposium, MDS '13*, pages 5:1–5:6, New York, NY, USA. ACM.
- [Machiry et al., 2013a] Machiry, A., Tahiliani, R., and Naik, M. (2013a). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 224–234, New York, NY, USA. ACM.

- [Machiry et al., 2013b] Machiry, A., Tahiliani, R., and Naik, M. (2013b). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 224–234, New York, NY, USA. ACM.
- [Maiya et al., 2016] Maiya, P., Gupta, R., Kanade, A., and Majumdar, R. (2016). Partial order reduction for event-driven multi-threaded programs. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, pages 680–697, New York, NY, USA. Springer-Verlag New York, Inc.
- [Maiya et al., 2014] Maiya, P., Kanade, A., and Majumdar, R. (2014). Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 316–325, New York, NY, USA. ACM.
- [Majumdar and Wang, 2015] Majumdar, R. and Wang, Z. (2015). Bbs: A phase-bounded model checker for asynchronous programs. In *International Conference on Computer Aided Verification*, pages 496–503. Springer.
- [Mirzaei et al., 2012] Mirzaei, N., Malek, S., Pasareanu, C. S., Esfahani, N., and Mahmood, R. (2012). Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5.
- [Musuvathi and Engler, 2004] Musuvathi, M. and Engler, D. R. (2004). Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI'04*, pages 12–12, Berkeley, CA, USA. USENIX Association.
- [Musuvathi and Qadeer, 2007] Musuvathi, M. and Qadeer, S. (2007). Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings*

of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pages 446–455, New York, NY, USA. ACM.

[Musuvathi et al., 2007] Musuvathi, M., Qadeer, S., and Ball, T. (2007). Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research.

[Mutlu et al., 2015] Mutlu, E., Tasiran, S., and Livshits, B. (2015). Detecting javascript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 381–392, New York, NY, USA. ACM.

[Netzer and Miller, 1990] Netzer, R. H. and Miller, B. P. (1990). On the complexity of event ordering for shared-memory parallel program executions. In *In Proceedings of the 1990 International Conference on Parallel Processing*, pages 93–97.

[Netzer and Miller, 1992] Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88.

[Ozkan et al., 2015] Ozkan, B. K., Emmi, M., and Tasiran, S. (2015). Systematic asynchrony bug exploration for android apps. In *International Conference on Computer Aided Verification*, pages 455–461. Springer.

[Papadimitriou, 1979] Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *J. ACM*, 26(4):631–653.

[Petrov et al., 2012] Petrov, B., Vechev, M., Sridharan, M., and Dolby, J. (2012). Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 251–262, New York, NY, USA. ACM.

- [Qadeer and Rehof, 2005] Qadeer, S. and Rehof, J. (2005). Context-bounded model checking of concurrent software. In *TACAS '05: Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 93–107. Springer.
- [Qadeer and Wu, 2004] Qadeer, S. and Wu, D. (2004). KISS: Keep it simple and sequential. In *PLDI '04: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24. ACM.
- [Rakamaric and Emmi, 2014] Rakamaric, Z. and Emmi, M. (2014). SMACK: Decoupling source language details from verifier implementations. In *CAV '14: Proc. 26th International Conference on Computer Aided Verification*, volume 8559 of *LNCS*, pages 106–113. Springer.
- [Ramalingam, 2000] Ramalingam, G. (2000). Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430.
- [Raychev et al., 2013] Raychev, V., Vechev, M., and Sridharan, M. (2013). Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13*, pages 151–166, New York, NY, USA. ACM.
- [Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). Artificial intelligence: a modern approach.
- [Sadowski et al., 2009] Sadowski, C., Freund, S. N., and Flanagan, C. (2009). Single-track: A dynamic determinism checker for multithreaded programs. In Castagna, G., editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 394–409. Springer.

- [Safi et al., 2015] Safi, G., Shahbazian, A., Halfond, W. G. J., and Medvidovic, N. (2015). Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 25–37, New York, NY, USA. ACM.
- [Savage et al., 1997] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP '97*, pages 27–37, New York, NY, USA. ACM.
- [Sen and Viswanathan, 2006] Sen, K. and Viswanathan, M. (2006). Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer.
- [Sinha et al., 2011] Sinha, A., Malik, S., Wang, C., and Gupta, A. (2011). Predicting serializability violations: Smt-based search vs. dpor-based search. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 95–114.
- [Steele, 1990] Steele, Jr., G. L. (1990). Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*, pages 218–231, New York, NY, USA. ACM.
- [Torre et al., 2012] Torre, S. L., Madhusudan, P., and Parlato, G. (2012). Sequentializing parameterized programs. In Bauer, S. S. and Raclet, J.-B., editors, *FIT*, volume 87 of *EPTCS*, pages 34–47.
- [Udupa et al., 2011] Udupa, A., Desai, A., and Rajamani, S. (2011). Depth bounded explicit-state model checking. In Groce, A. and Musuvathi, M., editors, *Model*

- Checking Software: 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, pages 57–74, Berlin, Heidelberg. Springer Berlin Heidelberg.
- [van der Merwe et al., 2012] van der Merwe, H., van der Merwe, B., and Visser, W. (2012). Verifying android applications using java pathfinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5.
- [von Praun and Gross, 2004] von Praun, C. and Gross, T. R. (2004). Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122.
- [Wang and Stoller, 2006] Wang, L. and Stoller, S. D. (2006). Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110.
- [Yi et al., 2012] Yi, J., Disney, T., Freund, S. N., and Flanagan, C. (2012). Cooperative types for controlling thread interference in java. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 232–242.