A Graph-based Theory to Analyze App Soundness Properties

Tooraj Helmi (thelmi@usc.edu)

Abstract

Apps are specific programs; however, the existing theories used with programs are insufficient to study app. This paper justifies the need for a theory applicable to apps, introduces it, and uses it to assert app soundness properties.

Introduction

Apps are ubiquitous. In 2021 more than 4 billion people used apps on their smartphones. However, a theory on apps is missing in software engineering. Almost anybody can quickly - but not accurately - answer the question of “what is an app?”. However, if we ask this question from people engaged in software research, it could take some time and deliberation to develop a definition applicable to all the apps. The description provided on the internet does not help to go any further either. Wikipedia defines a “mobile app” as “a computer program or software application designed to run on a mobile device such as a phone, tablet, or watches.” This definition includes the OS itself and programs that run in the background as the app which is not what we usually conceive of an app. Although multiple models like MVC, MVVM are used to govern the architecture of commercial apps, they are not suited to conduct an analytical study of the app's properties.

Specifically, we would like to define a model that can help us answer the following questions:

1. How do we know if an app is responsive or user-friendly?
2. How do we know if an app is going to crash at run-time?
3. How do we know if an app satisfies the user’s needs?
4. How can we synthesize an app?
5. How can we find areas an app can be optimized for?

We believe fundamental, self-containing research focusing on the basic properties of apps - which we call an app theory - is missing. The lack of app theory has manifested itself in many ways; For example, an app sometimes crashes after its release. It is typically not easy to figure out the issue and fix it in practice - since the information provided to the developer in crash analytics tools is not enough to allow for a proper simulation of the situation. One can ask why the primary app development tools like Visual Studio, Android Studio, or XCode do not detect the possibility of crashes during development? Is it necessarily a run-time task to detect these issues? As we will show in this paper, the answer is no; However, the lack of such analysis goes back to the same core problem: the non-existence of a theory that can precisely describe an app’s static and dynamic structure.

The theory of programs has been studied for more than 50 years. Floyd Hoare provided a theoretical study on the correctness of a program in his seminal paper \cite{hoare1969axiomatic} in 1969. Since then, many other theories have been introduced to study many aspects of programs. For instance, program abstraction \cite{clarke2004predicate} has been introduced to disguise complex details of programs to make it possible to apply static analysis to detect major defects during the compile time. Program hyper-properties were introduced to study the program from a non-functional perspective \cite{lamport1977proving}. The concept of time was applied to the methodology for the formal specification, verification, and development of reactive programs using the tools of temporal logic \cite{pnueli1992temporal}. Program synthesizer has been built using search-based techniques like CEGIS \cite{solar2008program} and SYGUS \cite{alur2013syntax}. All of this research has had a significant impact on advancing the theory of programs.

While we can apply the general program theories to apps, an app has a specific characteristic differentiating it from other programs. That is the existence of the user at the center of the app definition. First, most apps are deterministic, i.e., they have a particular defined input when the program starts. Thus, in an app, a user introduces randomness as he interacts with the app. Second, the app specification provided as a description of user intent \cite{survey} is much more complex than the specification provided for simple programs. Third, while most programs are meant to terminate, apps are designed to run until the user explicitly decides to end them. Last, the scale of time is much slower than what is typically expected in programs.

In this paper, we begin by providing several key definitions of an app and app model. We will then use these definitions to lay the main contribution of this paper, which is a graph-based theory on apps. The theory consists of several proven corollaries and algorithms to address four critical aspects of an app: soundness, optimization, verification, and synthesis. This paper focuses on the first aspect and offers algorithms that can detect app issues or detect suboptimal app areas.

Problem

There is no systematic way to analyze apps beyond general approaches used for generic programs statically. However, as mentioned earlier, apps are significantly different from regular programs. Specifically, given an app, we would like to make predictions about the following app characteristics:

Soundness: the app will behave without any deviation from expectations.
Verification: the app satisfies the user's intent.
Optimization: the app is designed and implemented to provide the best possible outcome.

We also want to generate an app given the user intent, which we call **app synthesis**.

Methodology

We start with a basic definition of an app. Next, we use this basic definition to derive a more complex meaning called the app model that specifies what an app is made of. We will then introduce a representation, called an app graph, that helps us introduce our theories about the app.

App Definition

We define an app as “a program that allows users to interact with a computerized device.” This definition includes three main components:

1. First, an app is a specification of a program.
2. Second, a user should be able to interact with an app.
3. Third, an app should run on a computerized device.

These components can and will affect each other. For example, the device’s hardware defines possible user interactions like touch, voice command, face recognition, etc. Or the user interaction can trigger the execution of different proportions of the program.

App Model

An app model is a systemic definition of an app that specifies what an app is made of. We use the basic app definition to reason about the app model. An app model contains six specific components: data, model, API, event, and controller.

Data

There are two ways that an app can respond to user interactions. Stateless and stateful. A stateless app does not memorize anything about the user’s past interactions. For example, a mirror app does the same thing every time it is launched: it reflects the object in front of its camera regardless of how many other images it has shown in that past. However, most apps are stateful; they keep track of user interactions, including typed information, the last screen opened, etc. - to adapt to user behavior.

Stateful apps keep “data.” Data can be either stored locally or remotely. In most cases, the data is kept remotely to run the app on more than one device - for example, a user might need to run the app even if she loses the original device. Also, many apps allow multiple users to interact with each other, in which case the combined data should be kept somewhere outside yet accessible to all devices.

Model

While it is possible to have an app make a round trip call to retrieve remote data all the time, this makes the app very slow and not practical. Hence, apps keep a local version of the data to run the logic that determines how to react to user interaction. We call this local version of the data “the model.” The model can be identical or different from the data in terms of its structure. Still, it is often synchronized with data to ensure local changes are not lost and include remote changes.

API

Not all app functionality can or may run locally. For example, when the level of processing is beyond what the device can handle, or when there is a need to have access to large amounts of remote data, the processing has to take place on a remote server. Another case that the processing may not happen locally is dealing with sensitive or confidential data; In such cases, APIs call the remote logic and return the results to the client. APIs can be called synchronously, meaning the app has to wait for its response to continue, or asynchronously, meaning it can continue right away.

Event

Apps function by responding to some events that can be internal or external to the app. The internal event is some specific condition that is defined in the model. For example, an event can be raised when a particular parameter's value gets larger than a threshold. An external event includes user interactions like tapping a button or receiving a response for an API.

View

Most apps include a visual user interface called view. The view visualizes the model and also allows the user to interact with the app.

Controller

A controller is the brain of the app. It can receive events. It contains specific logic to alter models, make API calls, or renders the views. Note that it is possible to have views and models implement such functionality - in such cases, the controller is implicitly implemented inside the view and model.

MVCADE

We use the six components: model, view, controller, API, data, and event to define the app model. Figure 1 shows the block diagram of an app based on MVCADE.

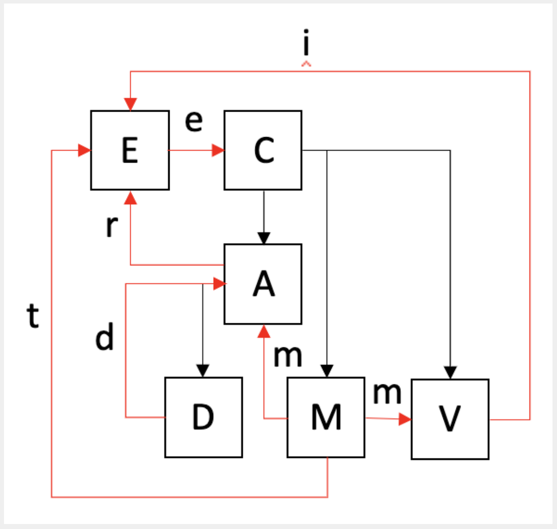


Figure 1. MVCADE App Model

App Graph

An app is typically made of several screens. Each screen typically has its view, model, and controller. At any given time, a screen is shown to the user, and based on the events, the controller of the current screen can decide to navigate to another screen, call an API, or update the model or view of the current screen. We use a graph to represent the dynamic behavior of the app. Figure 2 shows an example of the app graph.

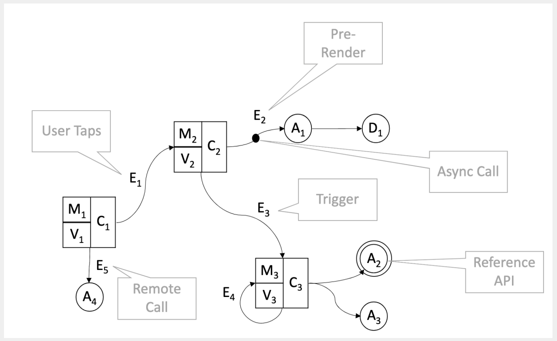


Figure 2. App Graph

As shown in Figure 2, there are different nodes in the graph. Each screen is represented as a rectangular node that contains M, V, and C for that particular screen. API and data are shown as A and D nodes. The edges represent navigation, API call, or a self-update. A label on the edge specifies the event. A dot is placed on an async API call, and a double-circle is used to indicate APIs that always return the same data - also known as reference APIs.

While all the components are required to explain how that app functions, some are used less frequently. Therefore we prefer to provide a simplified version of the graph that is shown in Figure 3.

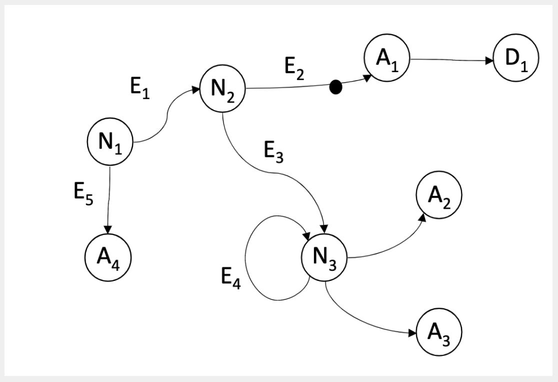


Figure 3. Simplified App Graph

In the simplified app graph, the screen components are lumped into a single node.

Graph-Specific Definitions

Using the simplified graph, we provide definitions that help us with the corollaries and algorithms provided in the following sections.

DEF 1.1. CORE NODE

A core node (N*) is a node that the user should return to using a bounded number of events.

DEF 1.2: NAVIGATION TIME

Navigation time between nodes (i) and (j) (t_ij) is the minimum time it takes to navigate from node (i) to node (j), assuming zero time spent on user input.

t_ij is the sum of the time for all none-async APIs on each node (k) [i ≤ k ≤ j] denoted as (a_k_r) where node (k) can have k_R APIs [0 ≤ k_r ≤ k_R]. Denoting the path from (i) to (j) as (l), we can calculate the navigation time as:

$$t_i^l = \sum_{k=i}^{j} \sum_{k^r=1}^{R} a_{k^r}$$

DEF 1.3: NAVIGATION EVENTS

Navigation events (e_ij) are the minimum number of events required to navigate from node (i) to node (j), excluding user input.

DEF 1.4: RETURNING LOOP

A returning loop is a path that starts from a core node and eventually returns to the same core node within a bounded number of events.

DEF 1.5: STICKING LOOP

A sticking loop is a path that starts from a core node and does not necessarily return to the same core node within a bounded number of events.

DEF 1.6: CONDENSED GRAPH, G*

A condensed graph is produced by eliminating all returning loops.

We can produce a condensed graph by applying algorithm 1.1 to a given simplified app graph.

```
Graph ProduceCondensedGraph(Graph g)
    for each $n_i$* is core nodes in g:
        $e_i = 0$, $t_i = 0$
        for each returning loop $l_k$ on $n_i$*
            $g = g - \{n \in l_k A\}$
            if ($t_i^l \geq t_i$ )
                $t_i^l = t_i$
            if ($e_i^l \geq e_i$ )
                $e_i^l = e_i$
    return g
```

ALGO 1.1. Generating Condensed Graph

App Soundness

Roughly speaking, an app is sound if it provides a satisfactory experience to its user. Unfortunately, we have all used not-sound apps Apps that crash, respond slowly to our interactions, or get stuck are some examples of such apps. In this section, we provide algorithms to measure and improve app soundness.

Non-Stickiness

In a sticky app, users can get stuck in some app screens, meaning they cannot get to other parts of the app once they get stuck. We would like to detect if such cases exist simply by analyzing the app graph. Collораries 1, 2, and 3 specify the conditions for an app to be sticky.

DEF 2.1: BLACKHOLE
A blackhole denoted as $\check{n}$ is a node in G* that, once the user enters, cannot get out of it.

DEF 2.2: MUTUAL BLACKHOLE SET
A mutual blackhole set, denoted by B, is a subset of nodes in G* that once the user enters, one will stay in that subset forever.

DEF 2.3: STICKINESS
An app is sticky if the user cannot reach any core node with a bounded number of events.

COR 2.1: CONDITION 1 FOR STICKINESS

$|G^*| \geq 2$ *and* $\exists \check{n} \in G^* \Rightarrow$ *App is sticky*

PRF:
A. if there is a blackhole and user never reaches it within a bounded numbe
events, then by DEF 2.2 app is sticky.
B. if user can reach the blackhole within a bounded number of events, then
2.1, it can never get out of it, which means that the other nodes cannot be r
so app is sticky.

$$A \wedge B \Rightarrow App\ is\ Sticky$$

COR 2.2: CONDITION 1 FOR STICKINESS
$|G^*| \geq 3$ *and* $\exists B \subset G^* \Rightarrow$ *App is sticky*

PRF:
A. if there is a $B \subset G^*$ and user never reaches any of the nodes in it, then by DEF 2.2 app is sticky.
B. if user can reach the any of the nodes in B then by def 2.3 is stays in that subset forever. This means that user can never any node not in B.

$$A \wedge B \Rightarrow App\ is\ Sticky$$

COR 2.3: NECESSARY AND SUFFICIENT CONDITION FOR STICKINESS

App is sticky $\Leftrightarrow$ ($|G^*| \geq 2$ *and* $\exists \check{n} \in G^*$) $\vee$
($|G^*| \geq 3$ *and* $\exists B \subset G^*$)

PRF:
A. If app is sticky then there is at least a node in G* that user cannot reach. Assume the rest of the nodes that user can reach are in $R \subset G^*$. If there is only one node is R then ($|G^*| \geq 2$ *and* $\exists \check{n} \in G^*$) *and if there are more nodes in R then R is a mutual blackhole set.*
B. Right to left is already proven by 2.1 and 2.

We can use ALGO 2.1 to assert if an app is sticky.

1. Generate condensed graph using ALGO 1.1
2. Use COL 2.3 to assert app stickiness

ALGO 2.1. Detecting app stickiness

Responsiveness

Being non-sticky is not enough to provide an excellent experience to the user. For instance, it can take a long time to navigate from a screen to another. This section introduces techniques to measure app responsiveness and offers methods to reduce the response time.

DEF 3.1: CRITICAL NODE
The critical node is a core node in a non-sticky G* with the highest navigation time.

$$N^C = \operatorname*{argmax}_{N_i^* \in C} t_{N_i^*}$$

DEF 3.2: RESPONSIVENESS
App is responsive ⇔ $t\_(N\^C\ ) < t\_R$

We can use ALGO 3.1 to assert app responsiveness.

1. Generate condensed graph using ALGO 1.1
2. Select the critical node - note that we obtain the navigation times as part of ALGO 1.1.
3. Apply DEF 3.2 to the asset if the app is responsive.

ALGO 3.1. Assertion on app responsiveness

DEF 3.3: PATH, SUCCESSOR, PREDECESSOR
If there is a path from node M to node N, N is called a successor to M with respect to path P, denoted as $M \leftrightarrow (P)\ N$. M is called the predecessor with respect to path P. (M ↔ N means a path from M to N, an inclusive set of nodes between M and N.

DEF 3.4: SUPERFLUOUS CALLS
A reference API (R-API) is superfluous if it is called more than once.

COR 3.5: VERIFYING SUPERFLUOUSNESS
An R-API is superfluous if there is more than one incoming edge to it.

DEF 2.14: INDIFFERENT CALLS
An API (A) is indifferent concerning M N, denoted as $M\leftrightarrow(A)\ N$ if calling A on M returns the same result as if it was called on N.

DEF 2.15: LATE CALLS
$M\leftrightarrow(A)\ N$ is a late call if A is called on node $P\in M\rightarrow N \wedge P\neq M$

User-Friendliness

A user-friendly app offers an effortless experience to the users. In this section, we introduce algorithms to measure app user-friendliness and techniques to improve friendliness.

DEF 4.1: FAR NODE
The far node is a core node with the highest navigation events.

$$N^F = \operatorname*{argmax}_{N_i^* \in C} e_{N_i^*}$$

DEF 4.2: USER FRIENDLINESS
App is user-friendly ⇔ $e\_(N\^F\ )< e\_R$.

We can use ALGO 4.1 to assert app user-friendliness.

1. Generate condensed graph using ALGO 1.1
2. Select the far node - note that we obtain the navigation events as part of ALGO 1.1.
3. Apply DEF 4.2 to the asset if the app is user-friendly.

ALGO 4.1. Assertion on app user-friendliness

Stability

Null pointer exceptions are the most significant cause of app crashes on Google Play[1]. This section introduces techniques to detect some of such crashes.

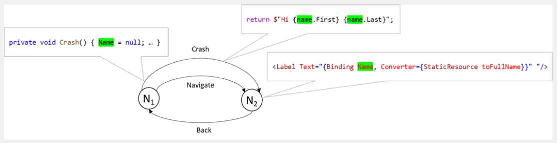
DEF 5.1: NULL-BOUND CRASH
A null-bound crash happens when at least one of its views has a binding to a property of a null object.

```
List<Assignment> DetectCrashTypeA(View v, Graph g)
    $A = \emptyset$
    for each binding $b \in v\ (b{:}\,control \rightarrow obj.prop)$
        $P$ = all incoming paths to $v \in g$
        for each path $p \in P$
            for each node $n \in p$
            $A^n$ = assignments which can set obj to null
                $A = A \cup A^n$
    return $A$
```

ALGO 5.1. Detecting null-bound crash

Results

We applied algorithm 5.1 to detect crashes in a mobile app developed in visual studio. The app is described in Appendix. While the app passes the validation included in major IDEs, it crashes when the user taps on a button. Using algorithm 5.1, we can detect the possibility of a crash during compilation. Figure 4 shows how we can detect a potential crash for this sample application.



```
return $"Hi {name.First} {name.Last}";
```

```
private void Crash() { Name = null; … }
```

```
<Label Text="{Binding Name, Converter={StaticResource toFullName}}" "/>
```

Figure 4. Detecting a crash in a mobile app.

[1] https://developer.android.com/games/optimize/crash

Conclusion and Future Work

In the paper, we introduced a graph-based theory on the app. Using this theory we were able to define soundness properties of the app based on the graph equivalent properties and derive algorithms that can detect and improve soundness issues including non-stickiness, responsiveness, user-friendliness, and stability.

While in this paper we only study the soundness properties, the graph-based theory has shown very promising results in our preliminary research on other categories of app properties and applications. These categories include: verification, synthesis, and optimizations. For example, we have synthesized “sound” apps based on some plain text requirements. While these requirements do not provide any information about what screens should exist or require the existence of navigational controls, our graph-based representation can make very useful assertions to generate a sound app.

References

@article{hoare1969axiomatic,
title={An axiomatic basis for computer programming},
author={Hoare, Charles Antony Richard},
journal={Communications of the ACM},
volume={12},
number={10},
pages={576--580},
year={1969},
publisher={ACM New York, NY, USA}
}

@article{clarke2004predicate,
title={Predicate abstraction of ANSI-C programs using SAT},
author={Clarke, Edmund and Kroening, Daniel and Sharygina, Natasha and Yorav, Karen},
journal={Formal Methods in System Design},
volume={25},
number={2},
pages={105--127},
year={2004},
publisher={Springer}
}

@article{lamport1977proving,
title={Proving the correctness of multiprocess programs},
author={Lamport, Leslie},
journal={IEEE transactions on software engineering},
number={2},
pages={125--143},
year={1977},
publisher={IEEE}
}

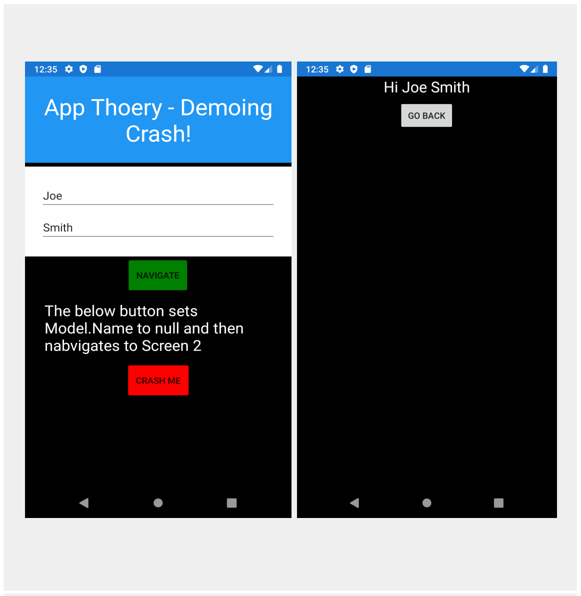
@article{pnueli1992temporal,
title={The temporal logic of reactive and concurrent systems},
author={Pnueli, Amir and Manna, Zohar},
journal={Springer},
volume={16},
pages={12},
year={1992},
publisher={Springer}
}

@book{solar2008program,
title={Program synthesis by sketching},
author={Solar-Lezama, Armando},
year={2008},
publisher={Citeseer}
}

@book{alur2013syntax,
title={Syntax-guided synthesis},
author={Alur, Rajeev and Bodik, Rastislav and Juniwal, Garvit and Martin, Milo MK and Raghothaman, Mukund and Seshia, Sanjit A and Singh, Rishabh and Solar-Lezama, Armando and Torlak, Emina and Udupa, Abhishek},
year={2013},
publisher={IEEE}
}

Appendix A. An example of a crashing app

The app shown in Figure 5 is a simple Android app consisting of two screens. There are two buttons on the first screen. The first one navigates the user to the second screen. The second button first sets the user object to null and then navigates. Tapping on the second button results in a crash since, in the second screen, we are binding the greeting sentence to the first name and last name properties of the user, which is null in this case. We have used Visual Studio to develop this app, but VS does not contain a tool that can detect the possibility of crashing even in this simple app.



The below button sets Model.Name to null and then nabvigates to Screen 2

Figure 5. The crashing app