# Developing Accessible Mobile Applications with Cross-Platform Development Frameworks

Sergio Mascetti
sergio.mascetti@unimi.it
Università degli Studi di Milano

Mattia Ducci
mattia.ducci@studenti.unimi.it
Università degli Studi di Milano

Niccoló Cantù
niccolo.cantu@studenti.unimi.it
Università degli Studi di Milano

Paolo Pecis
paolo.pecis@studenti.unimi.it
Università degli Studi di Milano

Dragan Ahmetovic
dragan.ahmetovic@unimi.it
Università degli Studi di Milano

## ABSTRACT

We illustrate our experience, gained over years of involvement in multiple research and commercial projects, in developing accessible mobile apps with cross-platform development frameworks (CPDF). These frameworks allow the developers to write the app code only once and run it on both iOS and Android. However, they have limited support for accessibility features, in particular for what concerns the interaction with the system screen reader.

To study the coverage of accessibility features in CPDFs, we first systematically analyze screen reader APIs available in native iOS and Android, and we examine whether and at what level the same functionalities are available in two popular CPDF: Xamarin and React Native. This analysis unveils that there are many functionalities shared between native iOS and Android APIs, but most of them are not available neither in React Native nor in Xamarin. In particular, not even all basic APIs are exposed by the examined CPDF. Accessing the unavailable APIs is still possible, but it requires additional effort by the developers who need to write platform-specific code in native APIs, hence partially negating the advantages of CPDF. To address this problem, we consider a representative set of native APIs that cannot be directly accessed from React Native and Xamarin and we report challenges encountered in accessing them.

## CCS CONCEPTS

• **Human-centered computing** → *UI toolkits*; *Accessibility*.

## KEYWORDS

Cross platform development, accessibility, mobile applications.

## 1 INTRODUCTION

When mobile developers write native code, they interact with the operating system (OS) through application program interfaces (APIs) exposed by the OS itself or by additional platform-specific development frameworks (which we label together as *system APIs*). Normally, each platform has its own set of system APIs, accessed through platform-specific programming languages. Therefore, applications intended for multiple platforms generally need to be developed separately for each different platform. Cross-platform development frameworks (CPDF) address this problem by wrapping system APIs for different platforms into their own APIs, which are consistent among different platforms and can be accessed through a single programming language. Thus, using CPDFs, developers can write their code only once and have working apps for both major mobile platforms (*iOS* and *Android*). Since CPDF wrap many of the commonly used system APIs, they have been shown to reduce projects' development and maintenance costs [1].

Our team (EveryWare Lab, University of Milan) has been developing mobile assistive technologies since 2010 and we recently started using cross-platform developing platforms to provide applications for iOs and Android. However, we soon realized that many native accessibility APIs (in particular related to screen reader) are not available through CPDFs APIs [3, 10], as confirmed by prior literature [15]. The developers can still access them, but they need to write *platform-specific code*, which means that the same function needs to be written separately for each target platform. Platform-specific implementations may differ in functionalities and user experience, which impacts the resulting app accessibility [6]. Furthermore, developing platform-specific code is time-consuming and requires the developers to know the native APIs, hence limiting the advantages of CPDFs themselves. As a result, prior works propose new mobile CPDFs, designed with accessibility in mind [9, 11] as previously done for desktop CPDFs [4, 7]. However, such approaches are not widespread and do not address the accessibility of apps developed with mainstream mobile CPDFs.

We first report a detailed comparison of mobile screen reader APIs, showing that iOS and Android screen reader APIs are functionally equivalent[1]. Then, we analyze how state-of-the-art mobile CPDFs wrap these native APIs. We show that *React Native* and *Xamarin*, two of the most adopted mobile CPDFs, only wrap about half and one third of the screen reader-related APIs shared by *iOS* and *Android*, respectively.

---

[1] *i.e.*, they expose the same (or very similar) functionality.

Even worse, the two considered CPDFs do not even expose all the *basic* accessibility APIs (a definition of *basic* accessibility API is provided in the following). Finally, we report our experience, discussing how we managed to access the APIs that are not wrapped by mobile CPDFs. Considering a representative set of APIs, we show examples for accessing them in *Xamarin* and *React Native*, with platform-specific code and we report how much effort this required.

## 2 BACKGROUND

### 2.1 Mobile Accessibility and Screen Readers

Mobile devices are accessed by people with disabilities through accessibility services that run in background and allow users to personalize how they interact with the device, for example providing a *magnifier* to people with low vision. In this experience report we focus on the challenges in developing mobile apps for people who are blind or have a severe visual impairment and use screen reader to interact with the device. Screen readers are accessibility services, available on both Android and iOS, that verbally describe elements accessed on the touch screen [8]. On *iOS*, *Voice Over* screen reader is part of the OS itself and third party solutions cannot be used. *Android*, instead, exposes APIs that enable third parties to develop screen readers. In practice, however, *Talk Back* screen reader, provided by Google, is most commonly used[2]. Hence, we only take into account *Talk Back* and *Voice Over*.

Screen readers can be used to access OS functionalities and apps, including those developed by a third party. In some cases an app can be (at least partially) accessible through a screen reader even though the app developers did not take explicit actions to enable screen reader accessibility. However, creating fully accessible apps often requires intervention from the developers. For example the developers have to specify the alternative text for images so that the screen reader can read them aloud. This form of intervention can be achieved though *screen reader APIs* that developers can use to interact with the screen reader or to personalize its behaviour. *VoiceOver* and *TalkBack* expose a number functionally equivalent APIs, which characterize fundamental non-visual interaction, but they also differ for a number of system-specific features, for example *VoiceOver* enables the use of a *rotor* gesture which is unavailable on *TalkBack*.

### 2.2 Cross-Platform Development Frameworks

Cross-platform development can be achieved with four approaches: web apps, hybrid apps, interpreted apps, and generated apps [14]. The first two approaches are based on web technologies and therefore may suffer from an overhead during interaction. Instead, in the latter two approaches, native code is automatically generated to create the user interface, presenting native *views*[3] to the user. Using native interfaces, these approaches yield better performance [13], thus improving user experience. Indeed they are the ones adopted by *React Native* and *Xamarin*, the two most popular mobile CPDF according to a survey conducted by StackOverflow[4], which we consider in the following.

---

[2] *Talk Back* is often pre-installed on *Android*, but it is not part of the OS.
[3] A *view* is an object shown on the screen with which the user can possibly interact.
[4] https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools

## 3 NATIVE SCREEN READER API

We conducted an analysis of screen reader APIs, considering our prior experience as well as the development documentation for *iOS* and *Android*. We created a taxonomy of the identified APIs based on the exposed functionality. Table 1 lists the 25 identified API functionalities and their availability in the four considered platforms (native *iOS*, native *Android*, *React Native*, *Xamarin*). Table 1 also indicates which APIs implement *basic* screen reader functionalities. To identify *basic* functionalities we took into account the accessibility principles and best practices presented in introductory accessibility documentation by Apple and Google [2, 5]. We considered the APIs mentioned in these resources as the *basic* ones.

The APIs are organized into the following five categories:

- **Accessibility focus**. A basic principle in mobile screen reader interaction is that the user can select a view, hear its description and then possibly activate it. When a view is selected, we say it receives the *accessibility focus*; at most one view can have the accessibility focus at the same time.
- **Text-to-announce**. When a view receives the accessibility focus, the screen reader reads aloud the textual description associated to it. This description, which we call *text-to-announce*, can be defined by the developers.
- **Explicit text-to-speech**. The text-to-announce is read aloud by text-to-speech (TTS) software. TTS can also be used by the developers to programmatically read a text aloud.
- **Accessibility tree**. Mobile screen readers define a logical tree structure to organize accessibility elements. The developers can access and edit this structure.
- **Miscellaneous**.

## 4 CPDF SCREEN READER API AVAILABILITY

The last two columns of Table 1 report whether each functionality is available in *React Native* or *Xamarin*, respectively. We denote that the functionality is available (**Y** symbol) if: (i) both *iOS* and *Android* expose a functionally equivalent API that is wrapped by the CPDF producing the same behaviour on both platforms, or (ii) an API is available in either *iOS* or *Android* and the CPDF wraps the API for that native platform, producing the expected behaviour in it and not in the other native platform. We indicate that a API is not exposed by a CPDF with the **N** symbol, while we use the **L** symbol to denote that the API is exposed by the CPDFs, but with a possibly different behaviour in the two native platforms.

In most cases (14 out of 25), *iOS* and *Android* implement functionally equivalent APIs (see Table 1). In all of these, it would be possible for CPDF to wrap native APIs into a single cross-platform API. In practice, however, this is not the case in both the examined CPDFs and, in particular in *Xamarin*. Indeed, out of the 14 functionalities shared by *iOS* and *Android*, only 8 and 5 are wrapped into *React Native* and *Xamarin* APIs, respectively. For the APIs that are exposed by only one native platform, it could still be possible for CPDF to wrap the API for that platform. Again, this is only rarely the case. Indeed, out of 11 API exposed by *iOS* or *Android* (but not both), only 5 and 2 are wrapped by *React Native* and *Xamarin*, respectively. The situation is only slightly better considering basic functionalities. Indeed, out of 8, only 6 and 5 are available in *React Native* and *Xamarin*, respectively.

**Table 1: Screen reader API functionalities and their availability in development platforms.**
**Notation: Y available; N not available; L available with limitations; basic according to *Android* (A) or iOS (I) documentation.**

| Category | ID | Basic in | API functionality | iOS | Android | React N. | Xamarin |
|---|---|---|---|---|---|---|---|
| Acc. focus | #1 | A | Specify which views should receive the accessibility focus | Y | Y | Y | Y |
| | #2 | I | Specify the accessibility focus order | Y | Y | N | Y |
| | #3 | | Assign the accessibility focus to a view | Y | Y | Y | N |
| | #4 | | Specify actions associated to accessibility focus-related events (e.g., toggle view focus) | Y | Y | N | N |
| | #5 | | Determine whether and which view has the accessibility focus | Y | Y | N | N |
| Text to ann. | #6 | A I | Specify attributes that contribute to form the text-to-announce | Y | Y | L | L |
| | #7 | | Programmatically define the text-to-announce | N | Y | N | N |
| | #8 | A | Use one view to describe another one | N | Y | N | Y |
| | #9 | | Specify that a view should be announced when changed, even without user interaction | N | Y | Y | N |
| | #10 | | Specify in which language the text-to-announce should be read | Y | Y | N | N |
| Explicit TTS | #11 | I | Read a text with the screen reader TTS | Y | Y | Y | N |
| | #12 | | Be informed when the screen reader finishes reading an explicitly provided text | Y | N | Y | N |
| | #13 | | Customize screen reader TTS speech features, like pitch, speed, etc... | Y | N | N | N |
| | #14 | | Read a text with non-screen reader TTS (also works when screen reader is not active) | Y | Y | Y | Y |
| | #15 | | Detect whether the non-screen reader TTS is reading | Y | Y | Y | N |
| | #16 | | Pause the non-screen reader TTS | Y | N | N | N |
| | #17 | | Customize non-screen reader TTS speech features, like pitch, speed, etc... | Y | Y | Y | Y |
| Acc. tree | #18 | A | Aggregate multiple views into a single accessible element | N | Y | Y | Y |
| | #19 | | Decompose a view into multiple accessibility elements | Y | N | N | N |
| | #20 | | Get the parent accessible element | Y | N | N | N |
| Miscellaneous | #21 | | Detect whether screen reader is active | Y | Y | Y | N |
| | #22 | A | Support navigation by specifying which views are headers or panes | N | Y | Y | N |
| | #23 | I | Define how to respond to screen reader user actions | Y | N | Y | N |
| | #24 | | Perform actions on user behalf | N | Y | N | N |
| | #25 | | Associate arbitrary accessibility-related information to a view | N | Y | N | N |

## 5   IMPLEMENTATION

We describe technical challenges in accessing native APIs not wrapped by CPDFs. For this, we implemented sample apps showcasing a subset of the APIs from Table 1, including all basic ones. The apps were developed in native code (*iOS* and *Android*) and in the considered CPDFs, deployed for both *iOS* and *Android*. The source code is available online[5], and the apps developed in *Xamarin* and *ReactNative* are published on Google Play Store and App Store[6].

Figure 1a shows the sample app for API #1, implemented in native *iOS*. The app shows three buttons that initially can receive the accessibility focus. Activating the first button results in the third button to become unfocusable. Figure 1b shows the sample app for API #2 implemented in native *Android*. Initially the accessibility focus order for the four presented buttons is the same as the visual order but when the first button is activated the accessibility focus order is changed. Figure 1c) shows the sample app for API #3 implemented in *React Native* and deployed to *Android*. The app shows three buttons: when the first one is activated, the accessibility focus is assigned to Button 3. Figure 1d shows the sample app for API #4 implemented in *Xamarin* and deployed to *iOS*. The app shows two buttons: when the second one receives the accessibility focus, it changes its color and label.

In Table 2, we classify the sample apps depending on how the accessibility functionality is implemented:

- *Direct API use*: the functionality is available as a platform API (**D** symbol);
- *Semi-direct API use*: the functionality is not available as a platform API, but a similar behaviour can be implemented using other platform APIs (**S** symbol);
- *Native API use* (for applications developed in CPDF only): the functionality is not available as a platform API and the implementation requires to access native APIs (**N** symbol).

For example, functionality #1 can be implemented with direct API use in all four platforms, as they all expose API for this. Instead, *iOS* does not expose an API for functionality #8, but it still can be implemented with semi-direct API use, using a combination of API #1 and #6 (both available in *iOS*). An example of native API use is functionality #4 in *React Native*: no API is available and it is not possible to implement the functionality using a combination other APIs available in the platform, hence it is necessary to use the native APIs available in *iOS* and *Android*. If an API is not available in a native platform, it also cannot be implemented in CPDF for that platform (denoted with an empty cell). For example, API #22 is not available in *iOS*, hence it cannot be implemented in *React Native* or *Xamarin* when deploying to *iOS*.

To give an idea of how challenging the development of accessible applications with CPDFs can be, in the following we report our
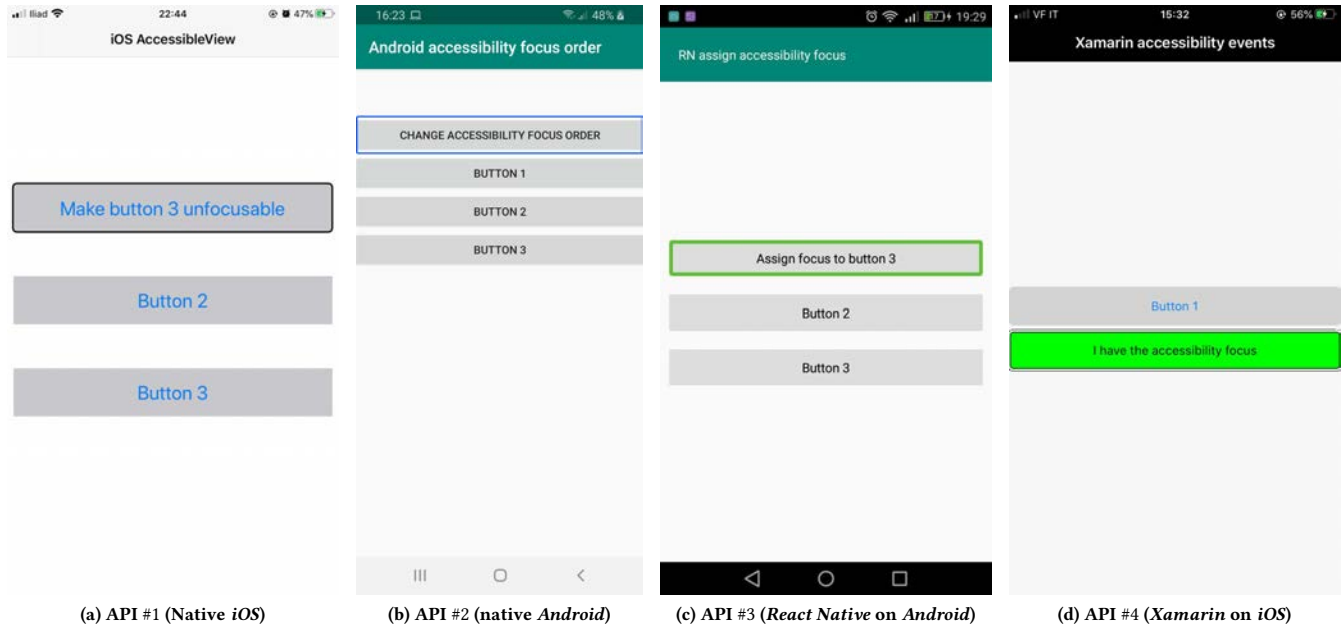
| (a) API #1 (Native *iOS*) | (b) API #2 (native *Android*) | (c) API #3 (*React Native* on *Android*) | (d) API #4 (*Xamarin* on *iOS*) |

**Figure 1: Examples of the samples applications**

**Table 2: Sample applications and how they are implemented (D = direct, S = semi-direct, N = native)**

| | Native | | *React Native* | | *Xamarin* | |
|---|---|---|---|---|---|---|
| **ID** | **iOS** | **Android** | **iOS** | **Android** | **iOS** | **Android** |
| #1 | D | D | D | D | D | D |
| #2 | D | D | N | N | D | D |
| #3 | D | D | D | D | N | N |
| #4 | D | D | N | N | N | N |
| #5 | D | D | | N | N | N |
| #6 | D | D | D | D | D | D |
| #8 | S | D | S | S | S | D |
| #11 | D | D | D | D | N | N |
| #14 | D | D | | S | D | D |
| #17 | D | D | | S | D | D |
| #18 | S | D | D | D | S | D |
| #21 | D | D | | D | N | N |
| #22 | | D | | D | | N |
| #23 | D | | D | | N | |

experience during the development of the sample apps. Considering the *Xamarin* implementation, as described in Table 2, we developed six sample apps based on native API use: #3, #4, #8, #11, #22, #23. All of them have been implemented using *dependency services*, except for the sample app developed for API #4, for which it was necessary to use *custom renderers*[7].

In both cases (*dependency services* and *custom renderers*) the greatest challenge for the developers is that they need to have the knowledge of both *Android* and *iOS* native APIs, their (sometimes

subtle) differences, and how they are wrapped into the *Xamarin C#* platform-specific APIs. This typically requires the developers to deal with the documentation for native *iOS*, native *Android*, *Xamarin.iOS* and *Xamarin.Android*. An additional challenge when using *custom renderers* is that the developers also needs to know how *Xamarin Forms renderers* work, and to be aware of all the classes involved in this part of the code.

To convey the amount of effort required for the development of accessible applications in *Xamarin*, we report, as an example, our own experience in developing the sample apps. The apps were implemented by a novice *Xamarin* developer (a co-author of this paper) supported by two professional developers, with 1 and 3 years of experience in *Xamarin* development, respectively, and a team of experts in mobile accessibility, with prior expertise in development of mobile assistive technologies on native platforms. For each sample app with direct API use about one hour of development time was needed, requiring limited support. The development time was similar for the applications based on semi-direct API use, but in some cases some support was needed by the team of experts in mobile accessibility to define how to combine exposed API to implement the expected behaviour.

Instead, sample apps with native API use required a much higher effort. In particular, those created using *dependency services* required about 2 hours of development each, and the intervention of the professional developers. Sample app for API #4, which required to use *custom renderers* required several weeks of investigation and despite the involvement of both supervising developers we were still unable to solve the problem. Eventually, we succeeded after receiving help from the official *Xamarin* support provided by a *Microsoft* developer.

---

[7]*Dependency services* and *custom renderers* are two technologies defined in *Xamarin* to access native APIs.

Considering the implementation in *React Native*, sample apps for API #2 and #4 were developed with native API use. This required, for each sample app, to develop a *native component*, which is composed of a few classes in native *Android* and *iOS*, hence written in *Java* and *ObjectiveC*, respectively. Each native component implements the accessibility functionalities by accessing native APIs and exposing them to the remaining of the *React Native* code. Similarly to what happens with *Xamarin*, the developers need to know the native accessibility APIs. However, in the case of *React Native* the developers also need to know how to code in native programming languages (*Java* and *ObjectivC*) and how the *React Native* native libraries work, and this is particularly challenging due to the fact that they are poorly documented.

Also in this case we report out experience with the development of the sample apps, created by one of the co-authors of this paper who is a professional developer with more than 3 years of experience in development with *React Native* and no previous experience in the development of native components. On average, he required less than one hour for each of the sample apps requiring direct or semi-direct API use. Instead, he required more than 10 days to complete sample app for API #4 on *Android* and he also needed to be supported by another experienced *iOS* developer to complete the *iOS* native component.

# 6 CONCLUSIONS AND FUTURE WORK

It is well known that many applications, including those for mobile devices, have accessibility issues [12]. In some cases, this is due to a lack of knowledge by the developers, who are not aware of the needs of people with disabilities, or do not know how to address these needs with the existing technology.

This paper uncovers a different problem: while native development platforms offer a broad range of APIs to develop accessible applications, many of these APIs are not wrapped by CPDFs. This means, for example, that a developer who is aware of the needs of people with visual impairments and who knows the native APIs to guarantee accessibility, will still struggle to implement accessibility functionalities in *React Native* and *Xamarin*. As we show in this paper, the developers can still implement these functionalities, but this require a higher development effort; in a commercial project this implies higher costs, hence it lowers the chances that the functionality is actually implemented.

Another relevant contribution of this paper is that it lists the accessibility functionalities exposed by *iOS* and *Android*. We do not have formal guarantees that this list is complete, but it represents a starting point for a discussion in the scientific and technical community. We believe that the list of accessibility APIs represents a valid technical reference documentation. It can be used by developers interested in creating accessible applications, including in particular mobile assistive technologies. Also, developers of new CPDFs and mobile OSs can refer to this paper to quickly understand which accessibility functions they need to implement.

This paper analyses an involved technical scenario and identifies a new relevant problem, hence paving the way for a number of follow-up research directions and technical interventions. On one side, this paper suggests that CPDFs can contribute to the development of applications presenting accessibility issues. This hypothesis can be verified in the future, for example with large scale studies, like the one conducted by Ross et al. [12]. The same methodology presented in this paper can be adapted to analyze the accessibility of other systems, for example traditional devices (desktop computers) and pervasive ones, or to address the accessibility to mobile devices by people with other forms of disabilities, like low vision. On the technical side, this paper identifies an intervention checklist for the development community to implement new functionalities in *React Native* and *Xamarin*. Both platforms are open source and hence it is possible to contribute in the development of new functionalities, which we intend to do. In the future we also intend to consider other CPDFs, including *Flutter*.

## REFERENCES

[1] Paulo Roberto Martins de Andrade, Otavio Frota, Fátima Silva, Adriano Albuquerque, and Robson Silveira. 2015. Cross Platform App: A Comparative Study. *Journal of Computer Science and Technology* (02 2015). https://doi.org/10.5121/ijcsit.2015.7104

[2] Apple. 2020. Apple Human Interface Guidelines. Accessibility section. https://apple.co/2VsZP37. Accessed: 2020-04-27.

[3] Niccolò Cantù, Mattia Ducci, Dragan Ahmetovic, Cristian Bernareggi, and Sergio Mascetti. 2018. MathMelodies 2: a Mobile Assistive Application for People with Visual Impairments Developed with React Native. In *ACM SIGACCESS Conference on Computers and Accessibility (ASSETS)*. ACM.

[4] Andres Gonzalez and Loretta Guarino Reid. 2005. Platform-independent accessibility api: Accessible document object model. In *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*. 63–71.

[5] Google. 2020. Android Developers Documentation. Principles for improving app accessibility. https://bit.ly/2K4gsg8. Accessed: 2020-04-27.

[6] Naomi Harrington, Yanyan Zhuang, Yağız Onat Yazır, Jennifer Baldwin, Yvonne Coady, and Sudhakar Ganti. 2013. Beyond user interfaces in mobile accessibility: Not just skin deep. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 322–329.

[7] Michael Heron, Vicki L Hanson, and Ian W Ricketts. 2013. ACCESS: a technical framework for adaptive accessibility support. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. 33–42.

[8] Shaun K Kane, Jeffrey P Bigham, and Jacob O Wobbrock. 2008. Slide rule: making mobile touch screens accessible to blind people using multi-touch interaction techniques. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*. 73–80.

[9] Peter Korn, Evangelos Bekiaris, and Maria Gemou. 2009. Towards open access accessibility everywhere: The ÆGIS concept. In *International Conference on Universal Access in Human-Computer Interaction*. Springer, 535–543.

[10] Sergio Mascetti, Giovanni Leontini, Cristian Bernareggi, and Dragan Ahmetovic. 2019. WordMelodies: supporting children with visual impairment in learning literacy. In *ACM SIGACCESS Conference on Computers and Accessibility*. ACM.

[11] Christoph Rieger, Daniel Lucrédio, Renata Pontin M Fortes, Herbert Kuchen, Felipe Dias, and Lianna Duarte. 2020. A model-driven approach to cross-platform development of accessible business apps. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 984–993.

[12] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2017. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. 2–11.

[13] Michiel Willocx, Jan Vossaert, and Vincent Naessens. 2015. A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services*. IEEE, 454–461.

[14] Spyros Xanthopoulos and Stelios Xinogalos. 2013. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*. 213–220.

[15] Yanyan Zhuang, Jennifer Baldwin, Laura Antunna, Yagiz Onat Yazir, Sudhakar Ganti, and Yvonne Coady. 2013. Tradeoffs in cross platform solutions for mobile assistive technology. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 330–335.