future internet

MDPI

Article

Supporting Elderly People by Ad Hoc Generated Mobile Applications Based on Vocal Interaction

Rita Francese *,† and Michele Risi †

Department of Informatics, University of Salerno, 84084 Fisciano SA, Italy; mrisi@unisa.it

* Correspondence: francese@unisa.it; Tel.: +39-89-963-316

† These authors contributed equally to this work.

Academic Editor: Georgios Kambourakis

Received: 20 June 2016; Accepted: 10 August 2016; Published: 25 August 2016

Abstract: Mobile devices can be exploited for enabling people to interact with Internet of Things (IoT) services. The MicroApp Generator [1] is a service-composition tool for supporting the generation of mobile applications directly on the mobile device. The user interacts with the generated app by using the traditional touch-based interaction. This kind of interaction often is not suitable for elderly and special needs people that cannot see or touch the screen. In this paper, we extend the MicroApp Generator with an interaction approach enabling a user to interact with the generated app only by using his voice, which can be very useful to let special needs people live at home. To this aim, once the mobile app has been generated and executed, the system analyses and describes the user interface, listens to the user speech and performs the associated actions. A preliminary analysis has been conducted to assess the user experience of the proposed approach by a sample composed of elderly users by using a questionnaire as a research instrument.

Keywords: service composition; Internet of Things; voice-based interaction; ambient intelligence; elderly people

1. Introduction

The elderly population is continuously growing and the World Health Organisation (WHO) foresees that its number will reach two billion by 2050. This population should live in the best way at home, in an autonomous way, well-being [2]. Smart home technologies are seen as a very promising support to in-home daily assistance by introducing automated control and assistive services [3] and by supporting context awareness (the capability of perceiving the surrounding physical environment and of adapting behavior accordingly). Multimodal interfaces are a key factor to improving the usability and accessibility of these systems to people with sensory limitations and elderly people [4,5]. In addition, the design of a smart home should depend on user requirements and living styles, and, consequently, has to be customized to the user [3].

The MicroApp Generator [1] is a system for generating mobile applications running directly on the mobile devices. It exploits a graphical service composition approach for composing applications starting from services that can be provided by the device (i.e., make a call to a specific number), web services available on the web (i.e., get the weather forecasts) or Internet of Things (IoT) services (i.e., turn off the air conditioner). In this paper, we extend the MicroApp Generator features that the caregiver of an elderly person may exploit to create for him an ambient assisting app that works also with vocal interaction. We provide some examples of MicroApps that can be exploited for supporting elderly people during their life at home, for their safety, including remote access, control features, and cross device notifications. We also let a sample of seventy elder people use the system and evaluate their user experience.

The rest of the paper is structured as follows: Section 2 discusses related work, while Section 3 summarizes the main MicroApp Generator features; Section 4 presents the vocal interaction extension of the MicroApp Generator; Section 5 reports the preliminary evaluation we conducted and, finally, Section 6 concludes the paper.

2. Related Work

Ambient intelligence provides healthcare facilities for elderly and healthy people. Each person has specific needs and lives in a specific environment and requires smart home applications customized on his/her lifestyles.

The automatic generation of mobile applications is a growing research area. Several proposals have been oriented towards end-user development exploiting service composition for general application [1,6–12] and mobile application generators for specific domains, including health and tourism/culture [13,14].

In [15], the development methodology for generating mobile applications exploiting mobile service composition has been evaluated through an empirical analysis that revealed that, in spite of the reduced size of the screen, the use of the MicroApp Generator tool improves the effectiveness in terms of time and editing errors with respect to the use of the MIT App Inventor [6]. App Inventor adopts a jigsaw programming approach and provides support for Web services. It is composed of two tools: *(i)* the *Designer*, a Web application that enables the user to select the widgets for the user interface; and *(ii)* the *Blocks Editor* running on PCs, for the visual programming language OpenBlocks [16]. Support to speech (i.e., speech-to-text, and text-to-speech) is provided through suitable block components.

Microsoft TouchDevelop [7] is a programming environment running on smartphones. It offers built-in primitives, making it easy to get the data provided by the mobile device sensors and by the cloud (via services, storage, computing, and social networks). The language is not graphical, variables and assignment statements are used for programming. Support to speech (speech-to-text, text-to-speech) is provided. The code has to be inserted by the developer. In our case, the vocal interface is automatically generated.

The "If This Then That" system, IFTTT [12] enables the user to use icons for generating simple apps based on triggering and actions (e.g., *"When I was tagged in a picture send me an alert"*). IFTTT has been integrated with Apple Siri. It is possible to execute IFFTT apps by pronouncing "hashtag" before the app name.

Atooma [11] is a mobile application running on both Android and iOS which allows the user to create basic applications performing one or more simple commands when a specific triggering event occurs. The triggering components have been developed for a variety of contexts, including ambient intelligence and, more detailedly, it supports safety and cross device notifications. Speech functionalities are available and have to be specified by the programmer.

Uday et al. [17] present a system that allows browsing the Internet by using a standard voice only, with the development of a Vocal User Interface. The browse takes vocal commands as input, translates them into Hypertext Transfer Protocol HTTP requests to the web server that processes it. The HTTP response is then speeched to the user.

In order to allow an effective comparison, we classify the main features of each tool in Table 1 depending on the following characteristics:

- *Components.* The kind of components that the tool can use for generating apps are classified as follows: *Service Components* (SC) to access Web and Internet services; *Sensor and Domotics Components* (SDC) to handle sensor data and networks; *Native Components* (NC) to exploit the functionalities available on the mobile device (e.g., phone call, camera, etc.).
- *Language.* The tool uses one of the following interaction metaphors to specify the applications: *Visual* (Vis), the user interacts by means of a visual/graphical language; *Template-Based* (TB), the user interacts by exploiting predefined forms; *Template-Based and Textual* (TBT), only simple apps can be programmed by using the template-based metaphor, whilst the others need textual programming.

- *Target Users.* It specifies if the tool is *End-User* (EU) oriented (i.e., no programming skills are required); or *Developer* (D) oriented (i.e., programming skills are required).
- *Target Device.* The final execution device on which the generated application will run: *Smartphone* (Sm); *Personal Computer* (PC).
- *Vocal Interaction.* The tool provides support for vocal interaction: *Text-To-Speech* (TTS) and/or *Speech-To-Speech* (STT) modules; *Screen* (Sc), the application is able to speech the text present on the screen and it enables the user to interact with the interface widgets.

Table 1. Technological features of related works and tools.

Tool	Components	Language	Target Users	Target Device	Vocal Interaction
App Inventor	SC, NC	Vis	EU	Sm	TTS, STT
Microsoft TouchDevelop	SC, NC	TBT	D	Sm	TTS, STT
IFTTT	SC, SDC, NC	Vis	EU	Sm	TTS
Atooma	SC, NC	TB	EU	Sm	TTS
Vocal User Interface	-	-	EU	Pc	Sc
MicroApp Generator	SC, SDC, NC	Vis	EU	Sm	TTS, STT, Sc

Legenda: [-] *Components*: *Service Components* (SC), *Sensor and Domotics Components* (SDC), *Native Components* (NC); [-] *Language*: *Visual* (Vis), *Template-Based* (TB), *Template-Based and Textual* (TBT); [-] *Target Users*: *End-User* (EU), *Developer* (D); [-] *Target Device*: *Smartphone* (Sm), *Personal Computer* (PC); [-] *Vocal Interaction*: *Text-To-Speech* (TTS), *Speech-To-Speech* (STT), *Screen* (Sc).

The novelty of the system we propose is not in the voice technology on a mobile system, which has been developed by Google or by Apple, but in the end-user generation of a mobile application with a vocal interface. The end-user that composes the application only has to take charge of its logic design.

3. The MicroApp Generator

The MicroApp Generator [1] enables a user to create mobile applications exploiting a service composition approach. This means that its elements are icons representing services. The iconic symbols are depicted by rounded rectangles, as shown in Figure 1. Each service icon on its top depicts an image representing the category of object handled by the service (e.g., a numeric datum) or the action performed (e.g., a camera for indicating the action of taking a picture).

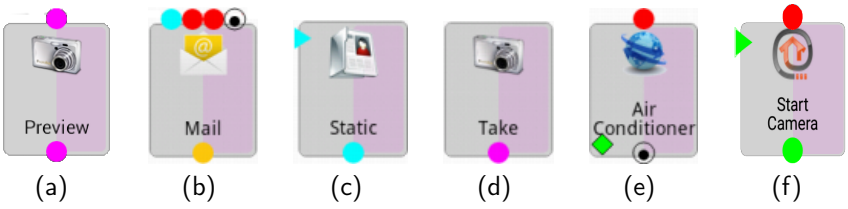


Figure 1. Some MicroApp service examples.

The MicroApp Generator is composed by a visual editor, which takes care of the definition and the modeling activities, and the MicroApp Engine, which is responsible for the execution activity, including the automatic generation of the Graphical User Interface (GUI) and the management of data exchange among services.

Services can be of the following types: *(i) device services*, such as *Phone.Call*, *Camera.Take*, and *SMS.Send*; *(ii) web services*, such as weather, book-plane, cypher-text; *(iii) home automation services*, such as starting the air-conditioner or opening the door. They are listed in the service catalog [18], the interface for the selection of the services to be composed. Figure 1 shows some examples of services offered by MicroApp Generator.

Input/output parameters are represented by colored circled dots on the top and on the bottom of the icon, respectively [19]. Colors represent the type of the parameter, e.g., pink corresponds to images, cyan to contacts, yellow to email objects composed of receiver, subject, attachments, and body. As an example, the leftmost service is the service *Camera.Preview* (see Figure 1a), which takes as input an image, displays it on the device screen, and provides it as output. Input parameters can be static, represented by triangles, or dynamic, represented by circles. The former are assigned at design time, while the latter at execution time.

In Figure 1b, *Mail.Send* receives as input a contact (represented by a cyan bullet) and two text strings (represented by two red bullets) for the subject and the body parameters of the email, respectively. The attached objects (represented by a black circled bullet) can be of any number and type and can be provided by different services. *Mail.Send* sends the email and provides it as output for possible printing, storing, etc. If multiple contacts are provided as input (implicit loop), the email is sent to each contact.

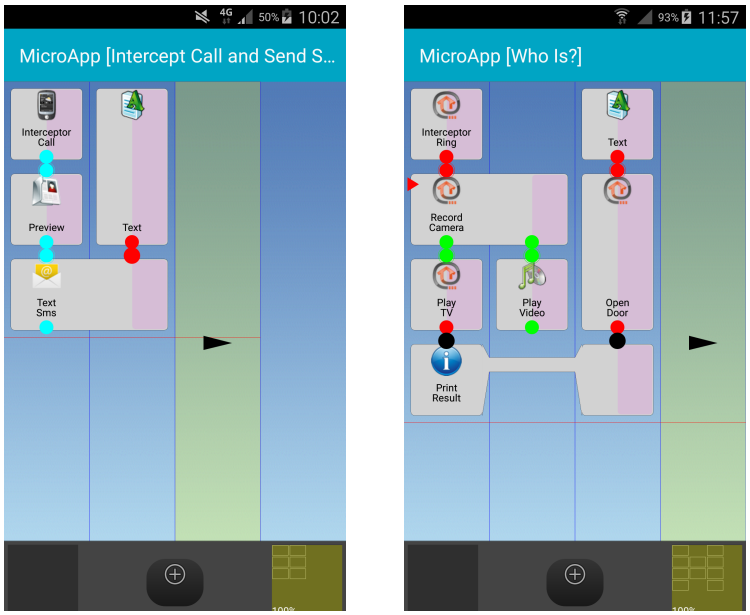
An example of a static parameter is shown in the *Contacts* service of Figure 1c. Once this static parameter has been defined, the system requires the user to select a contact from the contact list at design time. The user can also select more contacts, with the effect of producing a list of contacts that will be all managed automatically. For example, if this list is input to an email service, the same email will be sent to all the contacts in the list. Figure 1d shows the *Camera.Take* service, which has no input parameters. At execution time, the Image object provided as output is obtained. The MicroApp Generator also handles the native sensors (accelerometer, gyroscope, temperature, proximity and brightness) with specific services, similarly to how the GPS sensor is handled by the service *Location*.

The MicroApp Generator allows the user to define a constraint to be satisfied before starting the service execution (i.e., *pre-condition*). A pre-condition can be defined as mandatory or non-mandatory. A *mandatory* pre-condition, represented by a red diamond on the service representation, forces the application to stop if the pre-condition is not satisfied. A *non-mandatory* pre-condition, represented by a green diamond, enables the application to go on without executing the service, and the MicroApp Engine is in charge of defining a new control flow that excludes the services that are dependent on the stopped one. An example of using pre-conditions is shown in Figure 1e, where the service *AirConditioner.Set* takes as input the required temperature to be reached (i.e., 21 Celsius degrees) and it is executed in case the environmental temperature measured by the mobile device sensor satisfies the pre-condition (i.e., initial temperature to be higher than 26 Celsius degrees). Finally, Figure 1f shows a service that exploits a home automation service. In particular, the service *Camera.Start* is in charge to start a camera connected to the home automation system, which URI is provided by a static parameter, outputs the video-stream.

The MicroApp Generator offers context awareness features [1]. In particular, it collects the context information such as user position, current time, network connection, and environmental information. Depending on the user context, it detects the services available on the Web or in the user environment. In addition, Contingency Management is supported. The MicroApp Generator verifies if the involved services are available and tries to replace unavailable services, such as faults or network connectivity problems.

A MicroApp is designed by generating an application model, represented by an Extensible Markup Language XML file and constituted by a sequence of steps, where each step requires the execution of a single service. Each service exposes a description of its user interface that enables to generate automatically the MicroApp user interface.

Figure 2 shows two examples of app generated by the MicroApp Generator, useful for enhancing communication. The app in Figure 2a (namely *Intercept Call* and *Send SMS* lets a person that is unable to answer a phone call, to dictate some text and then send it as an SMS to the caller contact. The service *Phone.InterceptorCall* is a background service that intercepts an incoming call and provides to the preview service the contact info. Then, the service *Text.Text* takes in input a text by the user, which is provided as input to the service *SMS.Send*.



(a) (b)

Figure 2. Two MicroApp design examples: *Intercept Call and Send SMS* (**a**); *Who Is?* (**b**).

Who Is? is an example of cross-device notification. When the bell rings, the camera on the door is activated. The user can see who is ringing on the TV in the living room and let them in by triggering an event on the mobile device that is able to open the door. The application model is shown in Figure 2b. In particular, the *Ring.Interceptor* service detects the ring event and activates the camera installed on the home's door. The *Camera.Record* home automation service produces as output the video stream. Thus, the user can see who is ringing both on the TV in the living room (*TV.Play*) and also on the device screen (*Media.PlayVideo*). By pronouncing the "open" word, the user activates the *Door.Open* service. Finally, the result messages are spoken and displayed on the screen device.

The MicroApp model implicitly handles loops: each parameter represents a collection of objects of the same type. Thus, the data-flow of a MicroApp is represented by a directed acyclic graph. The vertices represent services and edges connections between the output and input parameters. Each service has a set of inputs provided by other services and, in turn, it provides inputs for other services. The service execution plan is automatically generated by constructing the topological order of the data-flow graph. This guarantees that, during the execution, all the data needed by a service are available. The XML model of a MicroApp is translated into a linear execution sequence by instantiating the service objects and running the process. A sequential mechanism enables the navigation forward and backward among the services when a MicroApp is running. Thus, the application can automatically, or after a user intervention, run the next service or come back to the previous one. Further details can be find in [1].

Another example of using Micro App Generator regards the creation of apps for supporting people with *memory impairments*, which affects the individual ability to accomplish common household tasks [20]. A caregiver can create a MicroApp composed of services representing the process of the activities to be conducted. As an example, the *use the washing machine* MicroApp may be composed of sequential steps needed to accomplish the whole washing process, starting from the charging of the machine, until collecting of the washed laundry. Each service registers the description of each step that is vocally communicated to the user. The user can listen to what to do and go forward and backward in the process, without the need of reading on the device. The program of the washing machine can be vocally selected by the user or prefixed by the caregiver, if the machine adopts smart technology.

4. Generating Apps with Vocal Interaction

In this section, we extend the MicroApp Generator tool proposed in [1] for enabling vocal interaction between the user and the generated application. In particular, the generated MicroApps have to *(i)* communicate with the user through vocal synthesis and *(ii)* the user will provide commands through voice by exploiting vocal recognition. This new feature enhances the accessibility of MicroApps, by enabling the user to interact with them also when he cannot physically access the device.

Figure 3. The Event (**a**) and Command (**b**) interactions between the user and the application.

4.1. Interaction Design

Each time the MicroApp requires user interaction, this has to be vocally performed by exploiting vocal recognition features. Anyway, the user has also to be able to interact by using the touch interaction modality. Vocal interaction may occur through:

- Vocal inputs, from the user. They correspond to the user interface events that the user needs to cause, such as pressing the "Next" button;
- Vocal outputs, from the MicroApp. They correspond to the description of the application interface, such as the availability of the "Next" button to be pressed or the possibility of inserting textual context into a text field with a given label.

Following these considerations, it is possible to divide the interactions between the user and the application in two main categories: *Event* and *Command* interaction. During an Event interaction, the MicroApp Generator reads the screen content to the user and asks him to provide vocal input, if some interface widgets need it, i.e., *"Pronounce a value"*. The user only has to pronounce the needed inputs. Figure 3a shows an example of Event interaction. During a Command interaction, the application exposes several commands that the user can execute, such as "Next" to go on the next widget, or "Delete" to remove the content of a text field. When the user pronounces one of the available commands, it is executed and the interaction goes ahead, as shown in Figure 3b. Generally, a Command interaction is followed by an Event one. The user interaction is accomplished through the following phases:

- *Screen widget acquisition.* To enable the interaction between the user and the MicroApp, there is the need of acquiring all the widgets composing the screen. The widgets are inserted into two stacks: the first is the stack of the uninterpreted widgets, which have not been read to the user yet, the other is the stack of the already read ones. Each time a widget is read, it is moved from the first stack into the second one. In the case of a command interaction, the user can go backward to previous read widgets, which are accordingly moved back into the uninterpreted widget stack. An example of the ordered acquisition of the *Camera.Take* service is shown in Figure 4, where widgets are ordered from high to low and left to right.

- *Screen reading.* The vocal interface gets the stack of the uninterpreted widgets and sequentially reads them, including the text they eventually contain. When a widget requires an user interaction, the interface communicates to the user the actions to be performed, i.e., insert text into a text field, and starts the voice recognition phase.
- *Voice recognition.* Once communicated the interaction, the vocal interface subsystem waits for the vocal input. It determines when the registration has to be interrupted by evaluating the time in which the user does not speak and a fixed maximum waiting time.
- *Command detection.* The user pronounces one of the available commands. Among them, it is always possible to navigate the widgets in a sequential way, by recalling commands to go to the successive or previous widget or for starting to read again the screen from the beginning. When the user input is not recognized, the system asks the user to insert it again.

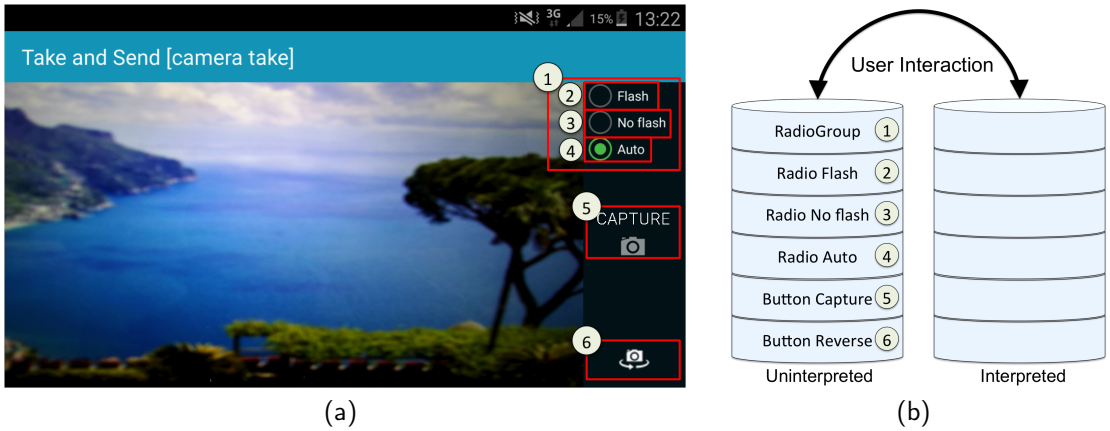


Figure 4. Two MicroApp design examples.

4.2. Implementation

MicroApps are implemented by following the *Template Method* design pattern, a behavioral pattern that offers a common logic to all the components and abstract methods, which have to be implemented in the classes extending it. Each service in a MicroApp is an extension of the MAActivity class.

All of the execution logic of the vocal user interface has been implemented at high level view in MAActivity. Thus, it is independent by the considered service.

Figure 5 shows the class diagram to implement MicroApp components. The activities of the various components (*CallInterceptActivity*, *SendSMSMessageActivity*, etc.) extend the MAActivity superclass which defines the basic structure and implements the Template Method design pattern. MAActivity uses the *SpeakAndSpeech* class, implementing *SpeechInterface*, listing the methods to be implemented for developing the vocal interface. In addition, the *SpeechAlertDialog*, *SpeechDatePickerDialog* and *SpeechProgressDialog* classes extend, respectively, *AlertDialog*, *DatePickerDialog* and *ProgressDialog*, and implement the *SpeechInterface* class.

The views in Android represent the widgets composing the visual interface of an application. A view is used to create interactive interface components in a rectangular area of the screen. It is responsible of the design and the event handling of a GUI component. All the views of a given screen are disposed in a tree. They act as controls or are used for displaying text, images and other content types. The present version of the vocal interface handles the views listed in Table 2, where the name of the supported view is reported together with its description and the need of providing vocal output and requiring vocal input.

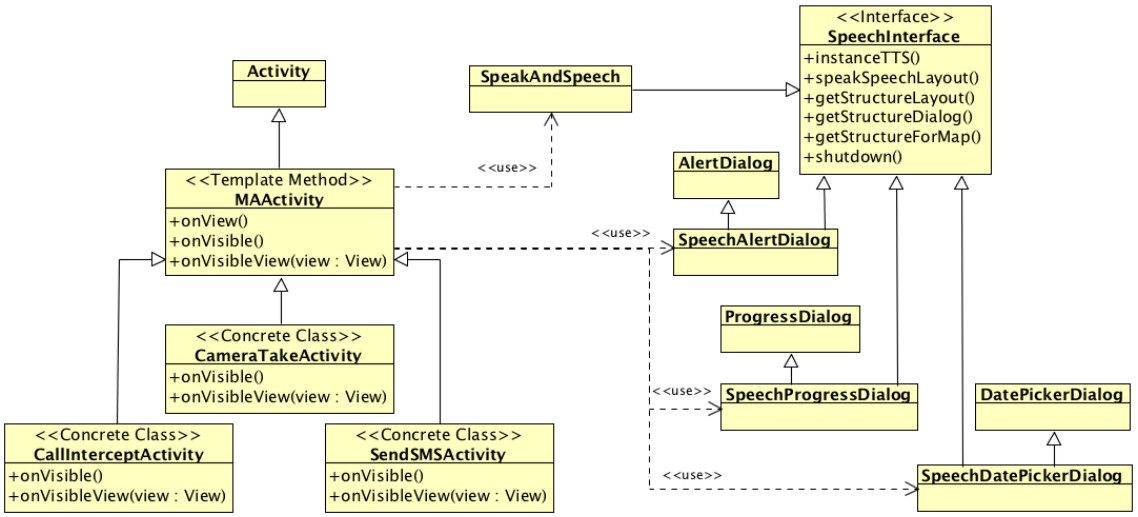


Figure 5. The Class Diagram of MicroApp components with speech capabilities.

Table 2. Android views handled by the MicroApp vocal use interface.

View Name	Description	Vocal Output	Vocal Input
TextView	Display text.	Y	N
EditText	Display text and enable to modify it.	Y	Y (not always)
Button	Button widget. Can be pressed to perform an action.	Y	Y
RadioButton	Two state button. It can be selected or unselected.	Y	Y (not always)
RadioGroup	Multiple exclusion for a set of radio buttons.	Y	Y
CheckBox	Two state button. It can be selected or unselected.	Y	Y (not always)
Spinner	Display one child at a time and lets the user pick among them.	Y	Y

4.3. The SpeakAndSpeech Class

This class is an intermediate between the service of a MicroApp and the vocal recognition features offered by the Android operating system. In particular, the *instanceTTS* method is responsible for initializing and running the vocal synthesis engine. This is an asynchronous operation to avoid delaying the application execution flow. Figure 6 shows the source code implementing the vocal synthesis engine initialization. In particular, during the initialization, the recognition/speech language is set to the one actually used on the device.

```
TextToSpeech tts;

public void instanceTTS() {
  tts = new TextToSpeech(..., new ListenerRecognition());
  ...
}

class ListenerRecognition implements RecognitionListener, OnInitListener {
  ...
  public void OnInit(int status) {
    if(status == TextToSpeech.SUCCESS) {
      int result = tts.setLanguage(Locale.getDefault()); //Set the language
      if(result == TextToSpeech.LANG_MISSING_DATA || result == TextToSpeech.LANG_NOT_SUPPORTED)
        Utils.debug("TTS or language is not supported");
    }
  }
}
```

Figure 6. The source code of the vocal synthesis engine initialization.

The reading of the interface of an activity begins inside the callback function communicating the starting of the vocal synthesis engine. The *getStructureLayout* method starts from the root of the actual screen views and inserts all the view children in the widget stack (see Figure 7). Initially, the widgets present in this stack are considered "uninterpreted". Similarly, the *getStructureDialog* and *getStructureForMap* handle dialogs and activities containing map widgets, respectively. Figure 8 shows the source code implementing how these methods are called starting from the screen view.

```
uninterpretedStack = new Stack<View>();
interpretedStack = new Stack<View>();

public void getStructureLayout(View view) {

  uninterpretedStack.addAll(getAllWidgets(view)); //Get recursively all the widgets from a view

  while(!uninterpretedStack.isEmpty()) {
    View v = uninterpretedStack.peek();

    if(!isLayoutWidget(v)) { //If it is not a layout widget (e.g., linearlayout)
      interpretedStack.push(uninterpretedStack.pop());
    } else {
      uninterpretedStack.pop();
    }
  }
}
```

Figure 7. The source code of method *getStructureLayout*.

```
  private SpeakAndSpeech sas;

    ...
    View v = onVisibleView(); //Get the view in the Activity
    if(v == null) { //Check if the view is a dialog
      sas.getStructureDialog();
    } else if (containsMap(v)) { //Check if the view contains map elements
      sas.getStructureForMap(v);
    } else {
      sas.getStructureLayout(v);
    }
  ...
```

Figure 8. The source code for handling the view layouts.

The *speakSpeechLayout* method acquires each view contained in the stack of the uninterpreted widgets and provides the text contained in each of them. The type of each view is identified for determining the listener to use in the vocal recognition callbacks (see Figure 9). The following listener types are considered:

- *null*: the view provides only text;
- *ListenerRecognitionEdit*: the view is a text field requiring the user to insert text;
- *ListenerRecognitionRadioGroup*: the view is a radio group containing a set of radio buttons requiring the user to select one of them;
- *ListenerRecognitionButton*: the view is a button requiring to be checked by the user.

The generated app will include the adequate listeners related to the interface structure.

Figure 10 shows the source code of the callback for handling the recognitions. In the case of unrecognized message, the system requires again to input one. For multiple recognitions, an alert dialog is displayed, and the user can choose one of the available matches or input a new one. The interaction with the alert dialog is performed vocally.

```
SpeechRecognizer sr = SpeechRecognizer.createSpeechRecognizer(...);

public void speakSpeechLayout(){
  while(!interpretedStack.isEmpty()) {
    View view=interpretedStack.peek();
    uninterpretedStack.push(interpretedStack.pop());

    if(view.getClass().getSimpleName().equalsIgnoreCase("TextView")) {
      TextView textView = (TextView) activity.findViewById(view.getId());
      String text = textView.getText().toString();
      if(textView.getLabelFor() != 0xffffffff) { //Check the widget associated to the TextView
        EditText editText = (EditText) activity.findViewById(textView.getLabelFor());

        if(!editText.getText().toString().equals(""))
          text = text + "; Value "+editText.getText()+"; Pronunce a value";
        else
          text= text + "; Pronunce a value";

        listenerRecognitionType = "ListenerRecognitionEdit";
        tts.speak(text, TextToSpeech.QUEUE_FLUSH, ...); //Speak the message
      } else {
        listenerRecognitionType = null;
        tts.speak(text, TextToSpeech.QUEUE_FLUSH, ...); //Speak the message
      }
    }
  }
  ...
}

...
Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);

if(listenerRecognitionType == null) {
  speakSpeechLayout();  //Handle the next widget in the view
} else if(listenerRecognitionType.equals("ListenerRecognitionEdit")) {
      sr.setRecognitionListener(new ListenerRecognitionEdit());
      sr.startListening(intent); //Get the value for the EditText
}
...
```

Figure 9. The source code of method *speakSpeechLayout*.

```
public void onResults(Bundle results) {

  ArrayList<String> matches = results.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);

  if(matches.size() == 0) {
    //User input is not recognized, insert again
  } else if(matches.size() == 1) {
    recognizedText = matches.get(0); //Recognized text
  } else if(matches.size() > 1) { //Handling multiple recognized texts
    SpeechAlertDialog.Builder builder = new SpeechAlertDialog.Builder(this);
    builder.setTitle("Pick a value");
    String[] vectorMatches = matches.toArray(new String[matches.size()]);

    builder.setItems(vectorMatches, new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) { //Called automatically by
                                                               //the performClick method
        recognizedText = matches.get(which);
      }
    });
  ...
  }
}
```

Figure 10. The source code for handling unrecognized, one or multiple recognitions.

In case the user pronounces a keyword (e.g., "Command") when the vocal recognition is listening, the system passes in the Command modality. The admissible commands enables the user to sequentially move among the screen widgets (e.g., "Up", "Down"), by exploiting their passage between the uninterpreted stack and the interpreted one.

4.4. An Example of Interaction

In the following, in detail, we describe, step-by-step, the interactions of the user with a MicroApp and, specifically the *Intercept Call and Send SMS* MicroApp whose design is shown in Figure 2a.

A MicroApp can be launched by the application menu or by the MicroApp Generator interface. In the second case, it is possible to use the vocal interface for selecting the application from the application list. The *Phone.InterceptorCall* GUI is shown to the user (see Figure 11a). When a call comes, the call service takes the control. When the ringing finishes, the *ContactPreview* service is shown and the interface content is described.

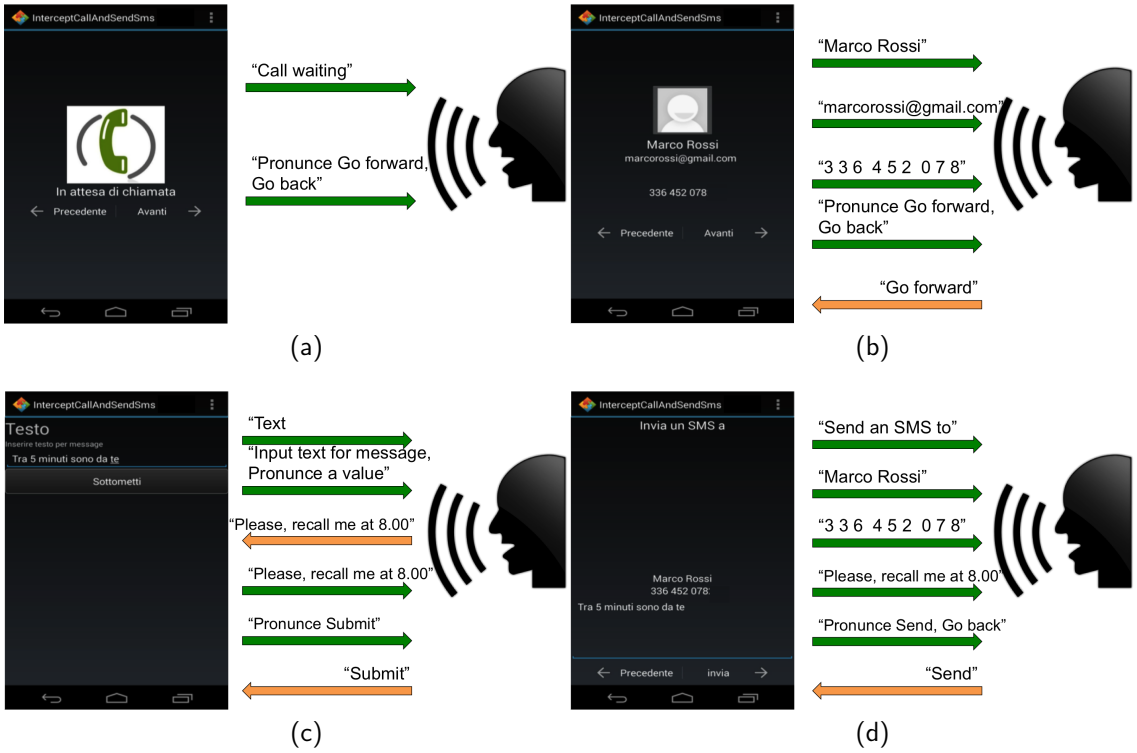


Figure 11. User interaction with the *Phone.InterceptorCall* (**a**); *Contact.Preview* (**b**); *Text.Text* (**c**); and *Text.SMS* (**d**) services.

When a call occurs, the *Contact.Preview* service is launched. Figure 11b shows the preview of the contact that performed the call (i.e., picture, name, surname, address and phone number). This figure also shows the vocal interaction with the user. When the user says the command "Go forward", the new interface related to the service "Text" is displayed (see Figure 11c). The interface asks the user to dictate the text of the message to be sent, e.g., "Please, recall me at 8.00". When the user says "Submit", the last service *Text.SMS* is executed (see Figure 11d). The vocal interface reads the text of the message and the contact data. At the end, the interface asks if it has to proceed or go backwards. If the user pronounces "Go forward", the service *SMS.Send* provides the data of the receiver and of the SMS text (Figure 11d). When the user says "Send", the SMS is transmitted.

5. Evaluation

In this section, we evaluated the user experience and explored the benefits and the pitfalls of the proposed vocal interaction approach.

5.1. Context and Procedure

We gathered and analyzed qualitative data from 16 people. The minimum age was 75 years old. We selected participants able to understand if the proposed technology could help them in case their autonomy was reduced. They had some difficulty in reading the device, they were people with low vision (their visual acuity of the best eye after correction is lower that 4/10).

After a little training of about ten minutes with the system, participants performed two tasks: *(i)* execute *Intercept Call And Send SMS*, depicted in Figure 2a and *(ii)* execute *Who Is?*, in Figure 2b. The evaluation was executed at the participant's home where the hardware was previously installed. During the tasks, participants were observed by a supervisor and, after completion, they filled in a questionnaire to evaluate the user experience.

We adopted the standardised user experience questionnaire of Laugwitz et al. [21], which evaluates the user experience in terms of *attractiveness* (Overall impression of the product. Do users like or dislike it?), *stimulation* (Is it exciting and motivating to use the product?), *perspicuity* (Is it easy to get familiar with the product?), *dependability* (Does the user feel in control of the interaction?), *novelty* (Is the product innovative and creative?) and *efficiency* (Can users solve their tasks with the product without unnecessary effort?).

Each factor's value represents the mean of four to six questions. Each question was rated on a seven-point Likert scale between opposites such as "annoying" (scored 1) and "enjoyable" (scored 7). An additional open question was proposed by the supervisor to participants for highlighting positive and negative points.

Regarding the preparation of the devices involved in the controlled experiments, we used a prototype supporting the composition of MicroApps on an Android based Samsung Galaxy S4 device, Android SDK version 5.0.1 [22]. Before starting the experiment to execute the mobile app *Who is?*, we installed in each participant's home the hardware architecture depicted in Figure 12. In particular, we used an OpenHab http://www.openhab.org/ server installed on the Raspberry https://www.raspberrypi.org/ device. The same device has also been connected to the TV, and to the doorbell and to the web camera we put on the door.

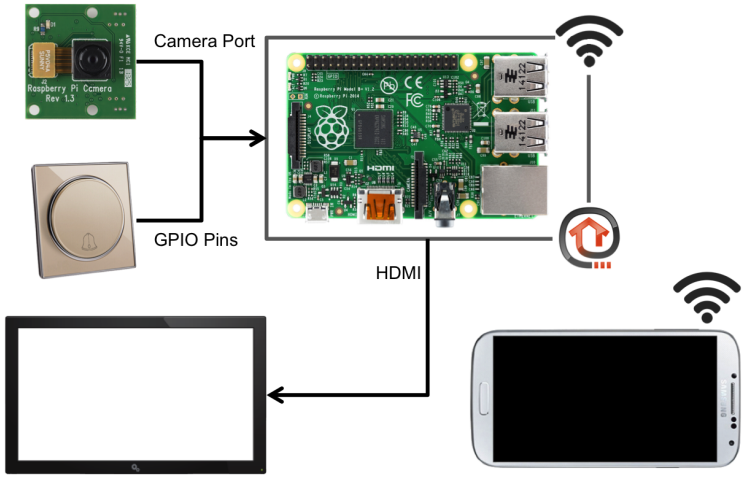


Figure 12. The hardware supporting the generated *"Who is?"* app.

5.2. Results

The user experience (UX) results are reported in Figure 13 as boxplots, which graphically depict groups of numerical data through their quartiles. We can note that most participants liked the system (*attractiveness*), only two scored three on average and four was neutral. All of the participants except two neutral were motivated to use the product (*stimulation*). Two people scored when rating *perspicuity*, so for them is not easy to become familiar with the product, but the users considered themselves able to control the interaction (*dependability*). They were favorably impressed by the novelty of the system that enables them to perform their task with a reasonable effort (*efficiency*).

The open questions provided the appreciation for possibility of controlling the environment by using voice. As a negative point, one participant did not like that sometimes he had to watch the

screen for accomplishing his tasks. One of the participants said: *"I liked very much to see the video on my big TV"*.

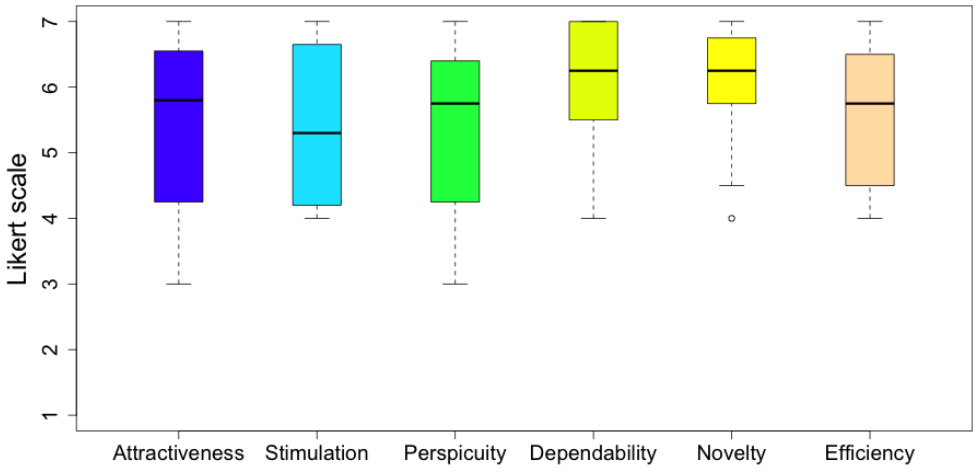


Figure 13. User eXperience (UX) results.

6. Conclusions

In this paper, we presented an extension of the MicroApp Generator that is able to create mobile applications whose interaction is based on voice. The MicroApp Generator with a voice recognition feature is a first step towards the customization of home automation services that take into account the user needs and life styles. The vocal interface reads the screen widgets to the user and reacts to his vocal input. This kind of applications can be useful for elderly people to exploit ambient intelligent features.

We also performed a preliminary evaluation on a sample of sixteen elderly people. First, results encourage us in designing other specific MicroApps, including apps for recognizing activity disorders such as falls, immobility, and empirically evaluating them. In addition, aged voice is often characterized by imprecise production of consonants, hoarseness and tremors. Moreover, in specific environments, the noise can affect the recognition performance and limits the usage of the developed solutions. To this aim, we also plan to evaluate how these problems may affect the system recognition by conducting controlled experiments in real and noised environments.

Acknowledgments: We thank all the experiment participants.

Author Contributions: Both the authors substantially contributed to all the parts of the papers.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Francese, R.; Risi, M.; Tortora, G.; Tucci, M. Visual Mobile Computing for Mobile End-Users. *IEEE Trans. Mob. Comput.* **2016**, *15*, 1033–1046.
2. Portet, F.; Vacher, M.; Golanski, C.; Roux, C.; Meillon, B. Design and Evaluation of a Smart Home Voice Interface for the Elderly: Acceptability and Objection Aspects. *Pers. Ubiquitous Comput.* **2013**, *17*, 127–144.
3. Alam, M.R.; Reaz, M.B.I.; Ali, M.A.M. A Review of Smart Homes—Past, Present, and Future. *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.* **2012**, *42*, 1190–1203.
4. Carbonell, N. Ambient Multimodality: Towards Advancing Computer Accessibility and Assisted Living. *Univers. Access Inf. Soc.* **2006**, *5*, 96–104.
5. Kolias, C.; Kolias, V.; Anagnostopoulos, I.; Kambourakis, G.; Kayafas, E. Design and implementation of a VoiceXML-driven wiki application for assistive environments on the web. *Pers. Ubiquitous Comput.* **2010**, *14*, 527–539.

6. App Inventor. MIT Center for Mobile Learning. Available online: http://appinventor.mit.edu/explore (accessed on 19 August 2016).
7. Tillmann, N.; Moskal, M.; de Halleux, J.; Fahndrich, M. TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen. In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Portland, OR, USA, 22–27 October 2011; pp. 49–60.
8. Wajid, U.; Namoun, A.; Mehandjiev, N. Alternative Representations for End User Composition of Service-Based Systems. In Proceedings of the Third International Symposium on End-User Development (IS-EUD), Torre Canne, Italy, 7–10 June 2011; pp. 53–66.
9. Ardito, C.; Francesca Costabile, M.; Desolda, G.; Lanzilotti, R.; Matera, M.; Piccinno, A.; Picozzi, M. User-driven Visual Composition of Service-based Interactive Spaces. *J. Vis. Lang. Comput.* **2014**, *25*, 278–296.
10. Aghaee, S.; Pautasso, C. End-User Development of Mashups with NaturalMash. *J. Vis. Lang. Comput* **2014**, *25*, 414–432.
11. Atooma. Available online: http://www.atooma.com (accessed on 10 August 2016).
12. IFTTT. Available online: http://ifttt.com (accessed on 10 August 2016).
13. Paschou, M.; Sakkopoulos, E.; Tsakalidis, A. easyHealthApps: e-Health Apps Dynamic Generation for Smartphones & Tablets. *J. Med. Syst.* **2013**, *37*, 1–12.
14. Sakkopoulos, E.; Paschou, M.; Panagis, Y.; Kanellopoulos, D.; Eftaxias, G.; Tsakalidis, A. e-souvenir appification: QoS web based media delivery for museum apps. *Electron. Commer. Res.* **2015**, *15*, 5–24.
15. De Lucia, A.; Francese, R.; Risi, M.; Tortora, G. Generating Applications Directly on the Mobile Device: An Empirical Evaluation. In Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI), Capri, Italy, 21–25 May 2012; pp. 640–647.
16. Roque, R.V. OpenBlocks: An Extendable Framework for Graphical Block Programming Systems. Master Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.
17. Uday, G.; Ketan, K.; Dipak, U.; Swapnil, N. Voice Based Internet Browser. *Int. J. Comput. Appl.* **2013**, *66*, 20–22.
18. Francese, R.; Risi, M.; Tortora, G. Management, Sharing and Reuse of Service-Based Mobile Applications. In Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), Florence, Italy, 16–17 May 2015; pp. 105–108.
19. Cuccurullo, S.; Francese, R.; Risi, M.; Tortora, G. MicroApps Development on Mobile Phones. In Proceedings of the Third International Symposium on End-User Development (IS-EUD), Torre Canne, Italy, 7–10 June 2011; pp. 289–294.
20. Mynatt, E.D.; Melenhorst, A.S.; Fisk, A.D.; Rogers, W.A. Aware Technologies for Aging in Place: Understanding User Needs and Attitudes. *IEEE Pervasive Comput.* **2004**, *3*, 36–41.
21. Laugwitz, B.; Held, T.; Schrepp, M. Construction and Evaluation of a User Experience Questionnaire. In Proceedings of the 4th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society on HCI and Usability for Education and Work (USAB), Graz, Austria, 20–21 November 2008; Springer-Verlag: Berlin, Germany, 2008; pp. 63–76.
22. Android 5.1 APIs. Available online: https://developer.android.com/about/versions/android-5.1.html (accessed on 19 August 2016).