Published in: IEEE TRANSACTIONS ON MOBILE COMPUTING journal.

The Version of Record is available online at: http://dx.doi.org/10.1109/TMC.2015.2422295

Visual Mobile Computing for Mobile End-Users

Rita Francese, Michele Risi, Genoveffa Tortora, *senior Member, IEEE,* and Maurizio Tucci

Abstract—We present an approach to enable the end-users to graphically compose their own applications directly on their mobile phone, mainly integrating the functionalities available on the device and those provided by pervasive and Internet services. To this aim, we propose a methodology and a graphical notation enabling the user to compose mobile applications, named MicroApps: the user creates an application following an incremental and iterative development process; he composes icons representing (pervasive) services mainly by touch-based selection and following a data-flow approach. He/she is not in charge of the creation of the user interface, which is automatically generated. The methodology enables the end-user to develop applications and/or compose services on the smartphone, so paving the way towards new scenarios where smartphones replace and overtake the Personal Computer, given their native possibility of wide connectivity, when augmented by features for interaction with remote systems and sensors. The methodology 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 MIT App Inventor [2].

Index Terms—Pervasive Mobile Applications, Graphical Environment, Service Composition.

1 Introduction

Recent advances in mobile technology, mobile networks and mobile computing offer new functionalities and applications for software systems on mobile devices. Their popularity is increasing also among users without specific technological skills.

Moreover, the demand for mobile applications comes from a wide range of domains: Gartner research expects a market volume of $185 billion in 2014 [40]. New services and innovative interaction modalities are continuously proposed, including gesture detection, device movement and context-based control [26]. These innovations are mainly due to the novel and cheap equipment offered by the latest generation of mobile phones, such as on-board cameras, accelerometers, compass, GPS, together with their increased processing power and fast Internet connectivity. In addition, around us there is an enormous number of instruments and sensors that need to be connected: "the Internet of Things" [3]. In this context, smartphones should allow the user to combine services across multiple instruments to get smarter applications [6].

Nevertheless, service composition is still a task for expert people since it requires the knowledge of complex standards and technologies, such as the standard executable languages WSCDL and BPEL. In addition, the development of mobile applications/services still requires the users to know specific programming languages (i.e., Java or Objective-C) and operating systems (i.e., Android, Symbian, iOS or Windows).

At present, mobile devices offer their functionalities through one of the following three modalities: *(i)* native applications, *ii)* services available on web sites, and *(iii)* applications that integrate native functionalities with predefined services (e.g., a camera application that enables to post a photo on the Facebook profile). On the other hand, the user may need to perform tasks that can be composed of several small steps of the previous modalities, e.g., *"take a picture, encrypt it and then send it to a predefined person"*. When a task of this kind is performed frequently, the user can take advantage of an application that automatizes it. The design and implementation of this kind of applications require programming skills that are uncommon among end-users.

In this paper we propose a mobile system, *MicroApp Generator*, to compose all kinds of services, ranging from web services and native smartphone applications to domotic services, i.e., services to manage highly sophisticated sensors and devices to control temperature, lighting, security systems, etc. MicroApp Generator lets the end-users compose pre-existing applications/services available on the smartphone, the local network and the web. Services are represented by rounded rectangles that can be connected to form more complex services, directly on the smartphone, following an incremental and iterative development process. It runs on smartphones equipped with Android SDK 4.2.2 or above.

In [11] and [12] we introduced the initial idea behind the MicroApp generation approach, based on graphical composition of functionalities of the mobile

• *The authors are with the Department of Management and Information Technology, University of Salerno, Fisciano, ITALY, 84084. E-mail: {francese, mrisi, tortora, tucci}@unisa.it*

device (i.e., phone call, camera, etc.). An initial prototype was evaluated in [17]. From this experimentation we understood the real impact on the Internet of Things and the need of studying a global visual environment to communicate with all kind of services, from domotic and sensor devices to web services and local smartphone applications. In the following we summarize the main contributions of this paper with respect to the results reported in [11] and [12].

MicroApp Generator enables the generation of pervasive services (i.e., MicroApps) through the composition of existing services directly on the smartphone by specifying a data-flow Direct Acyclic Graph (DAG) of services using sequential, fork and join compositions. It supports pre-conditions and loops in a transparent way and assists the user during the composition; MicroApp Generator does not require the user understand programming concepts such as assignments, variables, conditions, and loops. Conditionals are limited in the form of service pre-conditions.

The MicroApp programmer focuses just on the app behaviour, since the environment automatically creates a user interface for the app starting from the Web Service Description Language (WSDL) [9]. The MicroApp execution can be triggered by various conditions, including environmental and proximity ones, and gestures. If a service is unavailable at runtime, the tool will attempt to find another compatible service to replace it; MicroApps can be used as services in other MicroApps, supporting an incremental and iterative development process. The execution of a MicroApp is performed by interpreting a data-flow DAG of services represented as an XML description. This interpreter-based strategy makes it easy to test MicroApps while they are being created, and to load and execute MicroApps from the MicroAppStore, a shared MicroApp repository; thus, MicroApps can easily be shared with others and remixed to form new apps. The hardware required both to run the development environment (i.e., MicroApp Generator) and the generated application is a smartphone. All these design decisions are the results of trade-offs among the system simplicity, expressiveness and programming power.

To assess the usability of the proposed approach we conducted an evaluation to compare the effectiveness of MicroApp Generator on the smartphone with respect to the well-known PC-based MIT App Inventor. Even though the competition is unbalanced due to the wider facilities of a PC-based environment (keyboard, screen size) with respect to those available on the smartphone, the results of this investigation provide evidence that MicroApp Generator is better than App Inventor in some dimensions.

The paper is structured as follows: Section 2 describes the proposed system and the methodology underlying the MicroApp development environment. Section 3 describes an experiment to compare MicroApp Generator and MIT App Inventor with respect to usability, and Section 4 discusses its results. Section 5 analyzes the existing approaches in the field of end-user mobile application development and service composition. Finally, Section 6 concludes the paper.

2 End-user oriented mobile development in MicroApp Generator

In this section we present the philosophy of end-user oriented development using MicroApp Generator and its architecture.

Let us start by showing a sample application developed using MicroApp Generator. Consider the following scenario: Marc is a reporter of the newspaper *Daily News* in a war zone. Repeatedly, he sends to his editor by email encrypted pictures labelled by the name of the place where each picture has been taken.

The steps Marc performs to design the application *"Send to the Editor"* are depicted in Fig. 1, where services are represented by rounded rectangles and are connected through bullets which correspond to the service input/output parameters.

Services are classified by type of action or device sensor (i.e., Camera, Send, Facebook, Position) in the service catalog. To compose the application Marc performs the following actions: to take a picture, he selects the folder *Camera*, which collects all the services related to the device camera (*Camera.Take*, *Camera.Preview*, *Camera.Save*). By a first touch, he selects the service *Camera.Take* and, by a second touch, puts it in the first column (Fig. 1(a)). As he wants to see a preview of the picture before deciding to send it to his editor, he selects the service *Camera.Preview*. The output bullet of *Camera.Take* is compatible with the input bullet of *Camera.Preview* since both have the same color (pink), denoting the image data type. Then, to encrypt the picture, he first discovers the service *Encrypt* available on the web, and then puts it in the first column, since the output bullet of *Camera.Preview* is compatible with the generic input (black) of the selected service.

The editor's contact is selected by adding the service *Contact.Static* in the first available column on the right as shown in Fig. 1(b). Marc also has to send the picture location information. Thus, as shown in Fig. 1(c), he selects the service *Location*, which detects Marc's position (by means of the GPS of the smartphone), and puts this service in the third column. Next, he connects this service to the service *Maps*, which determines the name of the user location and produces a map of his position. With reference to Fig. 1(d), the service *Mail.Send* needs to collect the recipient email address, the picture and the location information. First Marc drags and drops the service *Mail.Send* in the first empty space of the first column attaching it to *Encrypt*. Successively he touches the

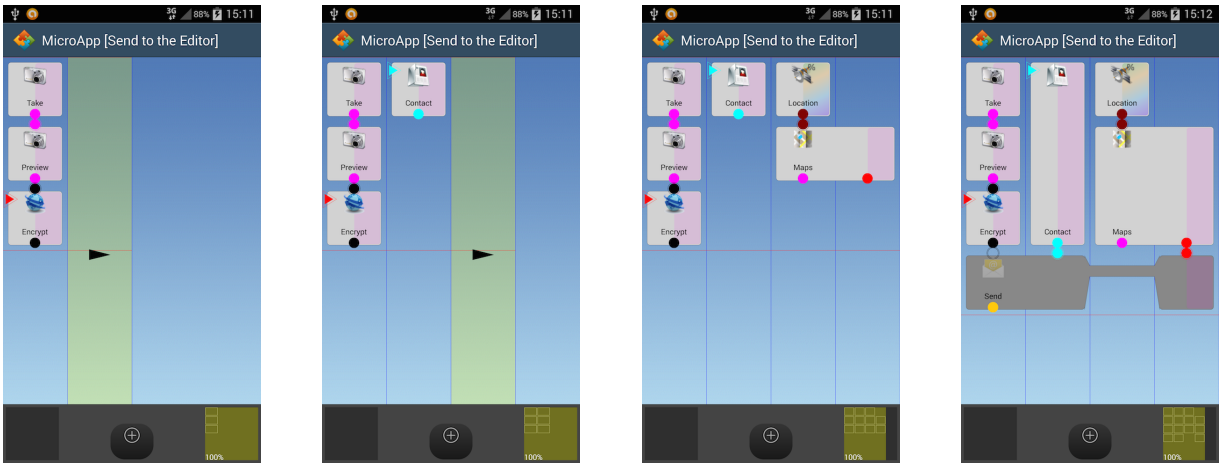


Fig. 1: The design steps of the application *"Send to the Editor"*.

services *Contact.Select* and *Maps* to associate their output parameters to the inputs of the service *Mail.Send*. In coupling parameters, the editor may automatically permute the input parameters in order to connect them correctly. Moreover, possible coupling ambiguities are solved by prompting a popup menu to the user. For example, *Mail.Send* has two parameters of type text (i.e., body and subject) both compatible with the location information outputted by Maps: from a popup menu, the user chooses to attach the location information to the body parameter. If there is an empty space between two services to be connected, the service icon is automatically lengthened, as in the case of *Contacts.Select* in the second column. Marc selects the editor's contact by longpressing on the service *Contact.Static*. A contact specified at design time (static) is set for all the successive uses of the application. He can specify more than one contact and the system will manage the collection of contacts in a transparent way. Similarly, Marc sets the password of the encryption algorithm for the service *Encrypt* at design time.

2.1 The composition process

The overall process to compose and execute a MicroApp mobile application follows an incremental and iterative approach. It consists of two phases: MicroApp Design and MicroApp Enactment, as shown by the UML activity diagram in Fig. 2.

MicroApp Design. The *Definition* activity allows users to describe the scaffolding of the MicroApp to be developed. The user chooses how the MicroApp is represented on the device. In particular, he provides the icon associated with the application and its name. The output of this activity is an empty MicroApp in XML notation. The MicroApp logic is composed considering the user context. Context-awareness represents the capability of a mobile system to perceive the surrounding physical environment and to

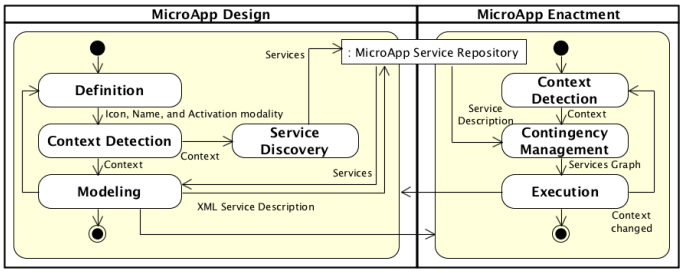


Fig. 2: The MicroApp development process.

adapt its behavior accordingly. Besides user's location context, several other factors, such as lighting, time, noise level, network connectivity, user actions and status, communication costs and bandwidth should be considered [5]. The context information is collected by the activity *Context Detection* and includes information such as user position, current time, network connection, and environmental information. Depending on the user context, the activity *Service Discovery* detects the services available on the web or in the user environment. If relevant services are detected, they are added to the *MicroApp Service Repository* and, once discovered, they can be reused anytime, anyplace.

A MicroApp is launched from the application menu of the smartphone. In addition, the user can select different activation modalities, by associating a user gesture (i.e., circle, a line from left to right, etc.) or by defining a trigger based on an environmental or proximity condition (*activation events*).

During the *MicroApp Modeling* activity, the MicroApp components available in the repository can be composed by connecting them, according to the input/output parameter constraints. The output of this activity is the XML description of both the static and dynamic aspects of the composed mobile application that is then stored in the MicroApp Service Repository on the device. In addition, the new

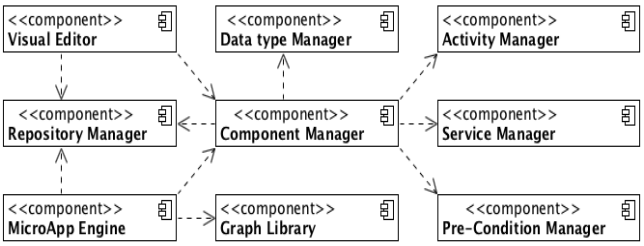


Fig. 3: The MicroApp Generator architecture.

MicroApp is registered in the action list launched when the selected activation event happens. Besides composing a MicroApp from scratch, the user can also download the design of an existing MicroApp from the *MicroAppStore* (i.e., a shared web repository of generated MicroApps) and modify it. This contributes to "Empower the end-users", changing their role from being passive recipients of services to active participants in service delivery and sharing [6].

MicroApp Enactment. Before executing a MicroApp, the activity *Context Detection* checks the user context. Then, the activity *Contingency Management* verifies the availability of the involved services and tries to replace unavailable services. In this way, the application is able to manage unpredictable availability of the involved services, i.e., faults or network connectivity problems. When all the required services are available, this activity provides as output a *Service Graph*, that represents the MicroApp design instantiated with the available services. The *Execution* activity linearizes the graph through a topological sort and starts the execution. The Execution activity periodically verifies if some context changes occur.

During the MicroApp Modeling activity the user can try his MicroApp at any time, by enacting it. In this way the development process is incremental and allows experimentation and testing of partially completed applications. This feature lets end-user skills grow gradually and provides immediate satisfaction.

2.2 The MicroApp Generator Tool

The MicroApp Generator architecture consists of the nine components depicted in Fig. 3.

The architecture includes two engines: the *Visual Editor*, which takes care of the definition and the modeling activities, and the *MicroApp Engine*, which is responsible of the Execution activity, including the automatic generation of the Graphical User Interface (GUI) and the management of data exchange among services. The engines get the description of the services from the *Repository Manager*, which is responsible of the MicroApp Repository on the user device, containing the XML description of the generated applications and its components. Moreover, the engines use the *Component Manager*, which handles the MicroApp services as components and associates the

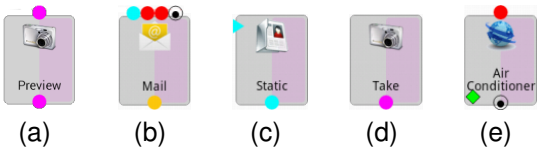


Fig. 4: Examples of service representation.

appropriate data type to the input/output parameters by exploiting the Data Type Manager. It also manages the parameter compatibility and their cardinality.

The engines use the *Activity Manager* to handle services corresponding to device functionalities (e.g., make a phone call or get the contact list) and the *Service Manager* to detect the user context, discover web and pervasive services, manage contingency and execute a remote service.

The MicroApp Engine uses the *Pre-Condition Manager* to verify whether the pre-conditions associated to each service are satisfied in the user context.

The *Graph Library* is called by the MicroApp Engine to linearize the execution flow of a MicroApp.

All these components are stored on the mobile device. An additional component is the *MicroAppStore Repository Manager*, hosted on an external server and responsible of the shared repository of the MicroApp user community. The applications developed by each user can be shared on the *MicroAppStore*. Since each generated application could contain user context information, such as contact telephone numbers or images, privacy should be protected when the application is shared with other users [20]. Thus, when transferring the design of a MicroApp from the device to the MicroAppStore, the Repository Manager removes all the private information, such as contact data or other static parameter values. A user can search and download a MicroApp of interest, customize it with his own data, and modify its behavior. This feature provides support to modification, since novice users often prefer to learn to use a tool by modifying an existing artifact. For example, user A creates the application model (the MicroApp XML description). Another user, B, downloads it from the MicroAppStore and runs it on his device, exploiting the native capabilities of the smartphone through Android. Thus, when B takes a picture, he will use the camera features offered by his device. A MicroApp accesses native sensors only by verifying their presence, at "sensor exists/does not exist" level.

2.3 Designing a MicroApp

The *MicroApp Visual Editor* is responsible of the Modeling activity of a MicroApp (see Fig. 2).

2.3.1 Service Representation

The basic elements of the composition approach are services, visually represented by rounded rectangles. A service is characterized by:

- a category icon, such as *Printer* or *Mail*;
- a service name, representing the action performed, such as *Make a report* or *Send*;
- the input parameters, represented by colored attaching points in the higher part of the rectangle;
- the output parameters, represented by colored attaching points in the lower part of the rectangle;
- a service pre-condition, which has to be verified before starting the service, represented as a diamond in the lower left side of the rectangle.

The MicroApp Visual Editor supports the definition of two kinds of parameters: *(i) dynamic parameters*, which are specified at run-time and represented by bullets; *(ii) static parameters*, which are assigned a value during the composition of the application. An end-user may create a new email service by setting at design time (statically) the email address of his frequently contacted friend; in this way he can use this application to send frequently emails to his friend without specifying his address every time. This value is set once for all the executions. Static parameters are represented by a colored triangle on the left hand side of the rectangle. A static parameter can be switched to become dynamic before the icon is used in the Composition Area. The choice between having a parameter static or dynamic is made at design time.

Concerning the *parameter cardinality*, each bullet or triangle represents one or more parameters of a given type, while a circled black bullet represents zero or more parameters of any type. Multiple instances of a parameter are handled by an implicit loop, which executes the service for each instance.

Fig. 4 shows samples services: the color of bullets and triangles represent parameter types. In Fig. 4(c) the cyan colored bullet represents a Contact object, containing the contact data (i.e., name, surname, address, email, phone numbers). In Fig. 4(a), the service *Camera.Preview* takes an Image object as input, displays it and returns it as output. In Fig. 4(b), *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 Fig. 4(c). Once this static parameter has been defined, the Visual Editor requires the user to select a contact from the contact list at design time. The user can also select more than one contact, 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. This feature is relevant for end-users, since they have generally some difficulties in managing loops [37]. Figure 4(d) 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* in Fig. 1.

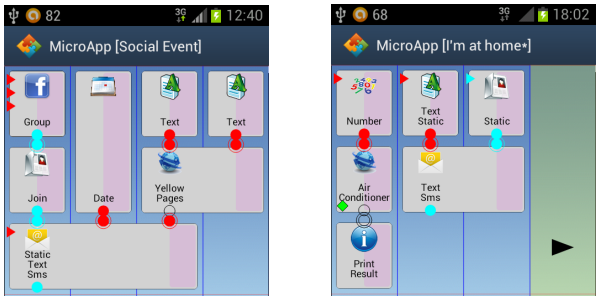
Since remembering the meaning of color parameters can be hard for a novice user, the editor provides a description of them when the user performs a long press on a service icon.

Pre-conditions are adopted to define a constraint to be satisfied before starting the service execution. Two kinds of pre-conditions can be defined: mandatory and non-mandatory. A *mandatory* pre-condition, represented by a red diamond, 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 use of pre-conditions is shown in Fig. 4(e), where the service *AirConditioner.Set* takes as input the required temperature to be reached (i.e., 21 Celsius degrees). By long pressing the service icon, a contextual menu is activated and the user selects the condition type (i.e., proximity, temperature). For example, he/she can set the value of the activation temperature to be higher than 26 Celsius degrees. The service is executed when the required conditions are verified by the user context. In this example, the air conditioner service is called in case the environmental temperature measured by the mobile device sensor satisfies the pre-condition.

2.3.2 Service Composition

The MicroApp Visual Editor adopts a data-flow programming approach, similarly to LabView [25] and ProGraph [10]. It offers a *Composition Area*, divided in rows and columns (see Fig. 1) where the user composes the application by dragging and dropping icons exploiting a touch-based interaction; this interaction modality eases the positioning of the service icons and the change of their position (low viscosity) [22].

The editor reduces *error proneness* [22]: it avoids errors in composing services by enabling the user to perform only the correct associations among data types according to the color of the parameter bullets. Colors also make evident the relationship among services, avoiding hidden dependencies that could affect comprehension [22]. In particular, two or more services can be composed if they are compatible. Given two services X and Y, an output parameter h of X is compatible with an input parameter k of Y iff h and k are of the same color or at least one of them is a circled black bullet. Two services in the



(a) *"Social event"* (b) *"I'm at home"*

Fig. 5: Examples of application design.

Composition Area are *disjoint* if they do not exchange any parameter.

End-users often make a series of sequential actions on the smartphone, such as take a picture, send it by email, etc. To create a sequence of operations the Visual Editor supports:

- *Sequential composition.* Two services X and Y are sequentially composed if they are in two successive rows and there exists at least one column of the Composition Area where the output parameter of X is compatible with an input parameter of Y.

The Visual Editor also supports parallel flow, offering the possibility of grouping together different services into one, instead of launching them sequentially. For example *"Send a picture to a friend both by email and MMS"*. The following two compositions are defined:

- *Fork composition.* It occurs on a service X when its output parameters are given in input to at least two disjoint services, as shown in Fig. 1(d), where the *Maps* service has one input and two outputs. The user can duplicate an output parameter to provide it to two or more different services. In this case, if the user long presses the output parameter bullet, the service icon is enlarged and the selected output parameter is duplicated.
- *Join composition.* It occurs when a service X receives inputs from two or more disjoint services (Fig. 1(d)).

The MicroApp Visual Editor avoids loop-like structures, since it automatically manages collections. The following examples show two applications that can be designed using MicroApp Generator. In particular, the application *"Social event"* composes web and social network services, while the application *"I'm at home!"* shows a sample SmartHome application using preconditions and pervasive services.

EXAMPLE 1.*"Social event"*. Alice often organizes meetings in public places, such as clubs or hotels, among the members of the Project Management Association. She needs an application that collects the data from the Facebook group of the association and sends an SMS with the venue information to the association members whose telephone numbers are stored in her smartphone. As shown in Fig. 5(a), Alice selects the Facebook service *Facebook.Group*, which provides a list of the group members as output; the service *Contacts.Join*, which fills the input contact list with the telephone numbers available on Alice's device; the service *Yellow Pages* providing information about the chosen venue, and the service *Send.StaticTextSMS* to send invitations to the selected contacts. In particular, the service *Facebook.Group* receives as input three textual static parameters: the Facebook login and password, and the group name. These parameters are specified once, at design time. The service *Yellow Pages* takes as input the search key and the place as dynamic text specified at runtime (service *Text*) and provides as output the textual information related to the chosen venue. The service *Send.StaticTextSMS* takes as input a static text (i.e., *"Project Management Association Meeting"*), the information related to the venue provided by the service *Yellow Pages*, the meeting date and sends their concatenation to the list of contacts as an SMS.

EXAMPLE 2.*"I'm at home!"*. Bob needs an application that tests whether he is close to his home and, if so : *(i)* it activates the air conditioner if the actual environmental temperature is higher than 26 Celsius degrees; *(ii)* it sends the message *"I'm at home!"* to his wife, who usually goes to her old father in the evening. Bob specifies the activation event, i.e., the following condition: *"he is at a distance less than a given radius from a predefined latitude/longitude position corresponding to his home"*. The application is activated when the location of the device falls into a circle of a given radius centered on Mark's home. To set the proximity radius Marc touches twice the map on the smartphone to set both his home location and the radius. The Visual Editor supports Bob in the composition of two execution flows, as shown in Fig. 5(b): *(i)* Bob sets the target temperature (e.g., 21 °C) of the AirConditioner by setting the static service *Number* to 21. Then he specifies the pre-condition of the service *AirConditioner.Set* by setting the activation temperature of the service to be greater than 26 °C, using the GUI shown in Fig. 6; a notification is sent by the service *Info.PrintResult*, which displays a *toast* message on Bob's device; *(ii)* two input parameters are provided to the *Send.TextSMS* service: the selected contact (i.e., Bob's wife) and the text of the message (e.g., "I'm at home!") to be sent to her (service *Text Static*), specified at design time.

2.3.3 Assisting the user composition

Since the system is devoted to end-users, the graphical notation and the Visual Editor have to reduce the cognitive effort, helping the user in resolving data type conflicts and data dependencies among services [28]. Examples of composition issues managed by the system are the following:

- *Automatic association.* Association between compatible service parameters is automatically resolved. The parameters are automatically permuted, simplifying

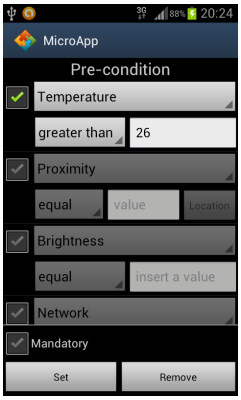


Fig. 6: The GUI for setting service pre-conditions.

the composition. In addition, the user can access information on the parameter type by a command in the contextual menu.

- *Avoiding erroneous input/output association*. The Visual Editor assists the user in case of ambiguous dependence among service parameters. When a user tries to associate services with different kinds of input/output parameters, the editor provides an error message and does not enable the association.

- *Bridge association*. The editor allows the user to connect the input parameters of a service with the output parameters of services that are not in contiguous columns. This case is shown in Fig. 1(d), where the Image object provided as output by the *Maps* service is not provided as input to the service *Mail.Send*.

- *Low viscosity* [22]. The editor lets change the position of an element in the Composition Area by touch-based drag and drop features.

- *Order independence* [22]. The composition can be performed selecting the services in different orders. Indeed, referring to Fig. 1(d), the user could first add the service *Mail.Send* and then position the other services. In particular, if there exists a service in the first row of the Composition Area that has at least one bullet input parameter, the Visual Editor automatically adds a new empty row by shifting vertically all the services by one position.

- *Combination* [22]. A more complex service can be created by combining existing services, including previously generated MicroApps that are managed in turn as services.

2.4 Service Management

MicroApps, in turn, can be used as services in other MicroApps, by means of their WSDL interface. Its logic is described following a specific XML-Schema stored in the MicroApp Service Repository. The adoption of a WSDL interface for specifying a MicroApp enables the tool to manage it as any other service based on Service Oriented Architecture (SOA). An example showing the recursive composition of a MicroApp is given in the Appendix.

To compose applications using both MicroApps stored in the MicroAppStore and pervasive/web services we adopt a discovery approach based on traditional techniques from the KNX standard [1] or the SOA field. The service catalog on the server contains a set of services periodically searched and validated. The user can search among them by specifying keywords matched with the Universal Description Discovery and Integration (UDDI) registry of the services. A ranked list of the services in decreasing order is generated adopting an Information Retrieval technique based on the Vector Space Model [4]. The user analyzes the list and selects the services of his interest, which will be available in the MicroApp Service Repository of the smartphone. Similarly, domotic services are integrated by exploiting the discovery functionality provided by the KNX protocol. If the user selects a previously developed MicroApp in the MicroAppStore, its XML description is downloaded.

The user gets detailed information on the service by long-pressing the service icon. He can also change the service name. When the user selects a web service, a MicroApp service is automatically generated and its icon appears in the service catalog on the device. The WSDL description of the service provides all the details useful to call it (service location and input/output parameters).

During the execution of a MicroApp, in case a web service is no longer available, the Service Manager tries to replace it automatically; it updates the service list by discovering new ones and searching an equivalent service whose input/output parameters correspond to those of the service to be substituted. If no substitute is available, the user is notified and the application is stopped.

The Service Manager invokes web services by using the SOAP client library *ksoap2* for the Android platform, whilst domotic services are invoked through the *Calimero* library.

2.5 Testing and Deploying a MicroApp

During the design activity the user tests the MicroApp by selecting the *Try* command. An XML description of the MicroApp is stored in the Repository, ready to be enacted by the MicroApp Engine, responsible of the MicroApp execution.

The data-flow of a MicroApp is represented by a directed acyclic graph since the MicroApp design does not use loops. Each service has a set of inputs generated by other services and, in turn, it provides inputs for other services. The service execution plan is automatically generated by a topological sort of the data-flow graph. The MicroApp Engine loads the XML description and translates it into a linear execution sequence by instantiating the service objects and running the process. As an example, Fig. 7(a) shows the directed acyclic graph computed on the

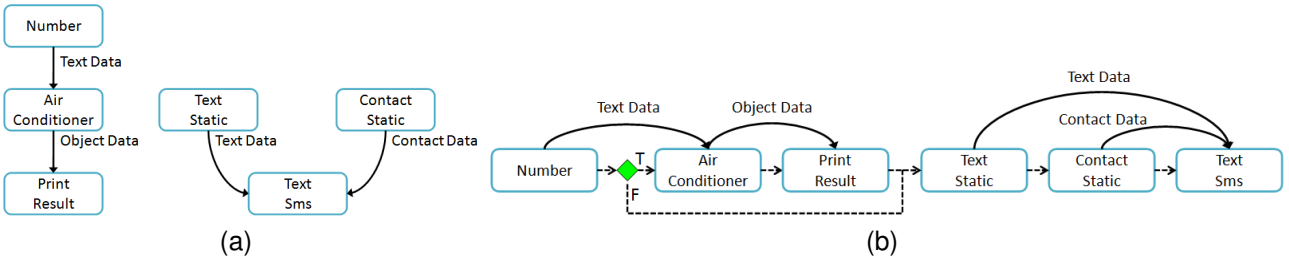


Fig. 7: The directed acyclic graph of a MicroApp (a) and its linearized execution plan (b).

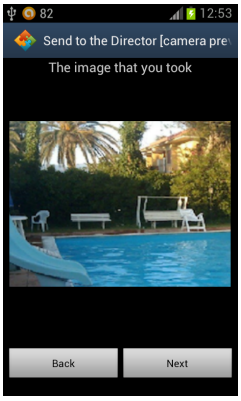


Fig. 8: The *Camera.Preview* user interface.

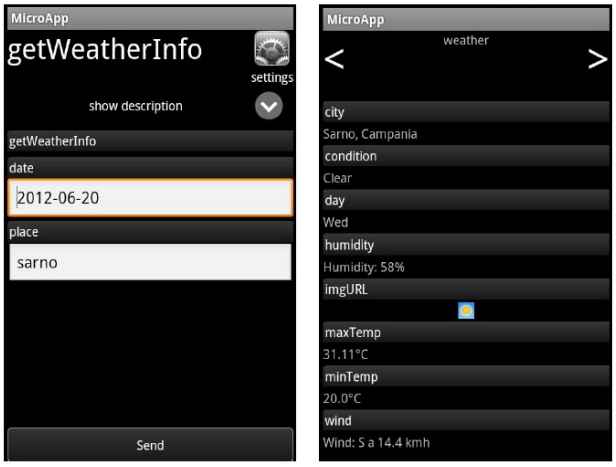


Fig. 9: Input and output interfaces generated for the getWeatherInfo web service.

MicroApp of Fig. 5(b): two connected components are generated starting from the XML description. The first one manages the air conditioning device and provides a feedback message as output, whereas the second one sends the SMS message to the specified contact. Figure 7(b) shows the linearized execution plan of the application. The solid arrows represent the data-flow, whilst the dotted lines describe the control flow. The pre-condition defined on the service *AirConditioner* is implemented as a decision node on the control flow. If the precondition is satisfied the service *AirConditioner* is executed, otherwise the control passes to the service *Text Static*. At the end of the modeling activity, the user selects the *Deploy* command to install the MicroApp on his device.

2.6 GUI Automatic Generation

The MicroApp Visual Editor assists the user in the composition of the application logic. There is also the need of modeling the GUI of the generated application in terms of windows, pull-down menus, buttons, scrolling, iconic images, wizards, etc. The composition of the user interface is not an easy task for end-users; thus, the solution adopted is the following: for native services, the GUI is generated using an XML description provided by the MicroApp Generator; for web services, it is automatically generated starting from the WSDL.

In our approach, each service interface is a slide, so a MicroApp has a slide-show presentation as interface [31]. Figure 8 shows the interface generated for the service *Camera.Preview*. A preview of the picture is shown to the user that can decide to accept it by touching the *Next* button or to move a step backward and take a new picture by touching the *Back* button.

The XML description of the user interface is dynamically created starting from the WSDL description of the operations and parameters of the considered service. In particular, MicroApp associates the appropriate graphical interfaces to the input/output parameters, depending on their data type, simple or structured. As an example, MicroApp Engine associates the Android widget *EditText* to an input parameter of type *String*, or a widget *Spinner* to a parameter of type *Enumeration*. Android widgets offer a standard user interface layout for which there is only the need of defining the content. As for output parameters, if the return value is URI, Image URI or String a *WebView*, *ImageView* or *TextView* interface is shown, respectively. The other return values use the interface *TextView* as default. The user interface of structured output parameters is generated recursively by using a collapsible widget.

Figure 9 shows the input and output interfaces of a web service that provides the weather forecast, named *getWeatherInfo*. The service requires as input the date and place of interest and provides the weather information.

2.7 Design decisions

In this section we discuss the design decisions, which conducted to the proposal of MicroApp Generator.

Our choices were the result of a trade-off between expressive power and user-friendliness.
- When designing the composition process we decided to foster the development by trials, which is appropriate for end-users [33]. To this aim we selected an incremental and iterative approach which enables a user to try his application, at any time, by enacting it.
- End-user development is also facilitated by the adoption of graphical metaphors [7] [29] [30] [38]. Thus, we decided to base the app generation on the composition of graphical icons.
- The Visual Editor has been designed taking into account the limited size of the device screen, then eliminating all the textual components and limiting the overall information displayed. In addition, the adoption of a touch-based interface simplifies the input mechanisms and minimizes typing. The choice of representing services by rounded rectangles simplifies the touch-based interaction, since they have been dimensioned considering the finger size.
- To simplify the composition of services and to connect them correctly we decided to adopt colors for representing the type of parameters. As it is generally agreed, the number of colors should be between 6 and 8 [22] and we followed this direction.
- Concerning the expressive power of the composition language we decided to adopt pre-conditions instead of conditions, because generally end-users think in terms of triggers such as *"in a given condition, perform this action"* [18], based on personal, spatial and temporal relationships (e.g., *"At ten o' clock, send a message to my wife"*).
- Implicit loops instead of loops were supported to handle any number of inputs in a transparent way. The number of input objects depends on the number of outputs produced by the input sources.
- Since loops are not admitted, it is possible to represent the service execution plan by an acyclic graph. The specification of the execution order of the activities is not an easy task for end-users [29]. Thus, we decided to automatically generate the service execution plan performing a topological sort of the data-flow graph, as detailed in Section 2.5. This guarantees that, during the execution, all the data needed by a service are available.
- Also the choice of automatically generating the user interface is the result of a trade-off between simplicity during the generation and user satisfaction of the generated application.

3 Evaluation

To assess the usability of MicroApp Generator we performed a user study comparing MicroApp Generator and App Inventor [2], the MIT tool generating mobile applications on the PC. App Inventor was initially developed at Google and now is managed by the MIT Center for Mobile Learning [2]. Generally, empirical studies carried out to compare the usefulness of two development environments/programming languages evaluate which one significantly reduces the errors and the time spent by the developer to complete a programming task. Thus, we measured Time and Error, since the lack of big screen and keyboard affects the time required and the error-proneness of the user interaction.

The primary purpose of App Inventor[1] is to democratize the creation of mobile apps. 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; *(ii)* the *Blocks Editor* running on PCs, for the visual programming language OpenBlocks [34]. It enables implementing applications that span the spectrum of mobile computing: stand-alone disconnected apps, multiperson games and other shared applications, clients for web services and databases, and interfaces to instruments and sensors.

An evaluation of the initial prototype of MicroApp Generator has been conducted in [17], where it has been compared to App Inventor Classic. That experiment was limited to the composition of functionalities of the mobile device. The evaluation proposed in this paper concerns the use of mobile computing features of both MicroApp and App Inventor related to the development of applications executing web services integrated with native device features, and exploiting the user context.

3.1 Experiment definition and context

The study was conducted in the Mobile Computing Lab of the University of Salerno. Data for the study have been gathered considering a group of 40 students in Computer Science that voluntarily took part in the experiment. The choice of Computer Science students as participants of an End-User development experiment could be a threat to the experiment validity. For this reason, we considered only Computer Science students at the beginning of their first year of University and, after a Pre-experiment questionnaire, we selected as participants only those having no programming skills.

During the experiments, we assigned the following two tasks to each participant, consisting of the development of small functionalities:
- T_1: implement and generate the mobile application *Stock Quotes*, selected from the tutorials proposed by App Inventor. In particular, the task requires to call a web service (i.e., *Yahoo! Finance*) to get the latest price for a stock, and to visualize the numeric result. Figures 10 and 11 show the composition of the application by using MicroApp Generator and App Inventor,

1. There exist two versions of App Inventor: App Inventor Classic and App Inventor 2. In this study we adopted the former.

TABLE 1: Experimental Design.

	Group 1	Group 2	Group 3	Group 4
Lab_1	T_1-M_{MA}	T_1-M_{AI}	T_2-M_{MA}	T_2-M_{AI}
Lab_2	T_2-M_{AI}	T_2-M_{MA}	T_1-M_{AI}	T_1-M_{MA}

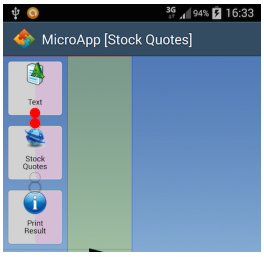


Fig. 10: The application *"Stock Quotes"* in MicroApp.

respectively. The App Inventor blocks to get stock quotes (Web.Url and Web.GotText) are generic blocks that access information to any Web page or Web service. In Fig. 11, the programmer has filled them in with the details that allow access to Yahoo finance stock quotes. In contrast, the Stock Quotes service icon in the MicroApp design represents a mechanism in which this information has already been filled in.

- T_2: implement and generate the mobile application *I'm at home*, described in Section 2.3.2 (the App Inventor solution is reported in the Appendix).

Two different methods, namely M_{MA} and M_{AI}, using MicroApp Generator and App Inventor, respectively, have been considered for the tasks T_1 and T_2.

Effectiveness is evaluated considering the dependent variables *Time* and *Error*. The former is measured as the time (expressed in minutes) needed to accomplish the task. *Error* measures the number of editing operations required to change the application provided as output by the participant into a correct one.

Mono-operation and mono-method biases are avoided thanks to the adoption of two tasks and two methods. In particular, the considered treatments are all combinations of the Methods (M_{MA} and M_{AI}) and Tasks (T_1 and T_2). To avoid results to be biased by group ability, each user experienced both Methods and both Tasks over the two subsequent laboratory sessions Lab_1 and Lab_2. Also, to minimize the learning effect, we needed to have users starting to work in Lab_1 using both the Methods (M_{MA} and M_{AI}) on both the generation tasks. Table 1 summarizes the design of the experiments, where T_i-M_j indicates the combination of task and method performed by a group of users in each laboratory session. To avoid the threats to validity due to history and maturation, the circumstances are the same in both the lab sessions and the two sessions occurred on the same day. The groups of students were homogeneously composed considering scores reported by them in the admission test for the degree program.

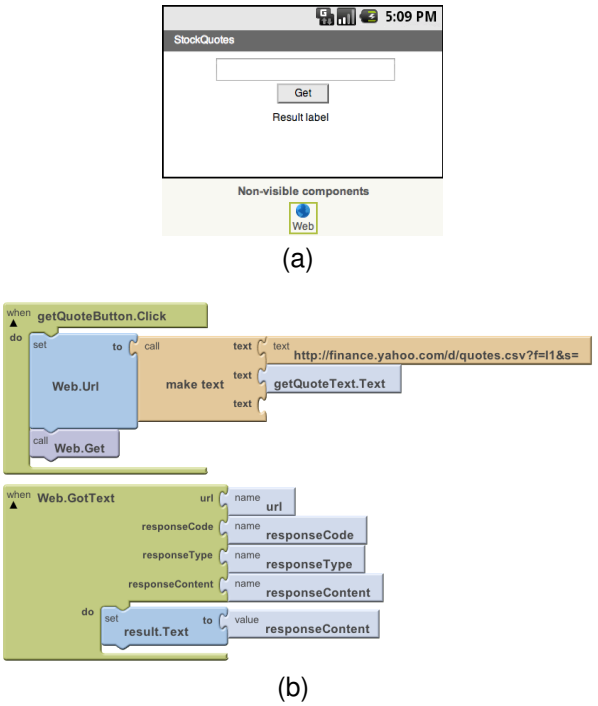


Fig. 11: The user interface (a) and the visual composition (b) of the application *"Stock Quotes"* in App Inventor.

3.2 Preparation, Material and Execution

The study has been divided in three steps and performed in one-to-one sessions (i.e., a supervisor for each subject) using the think aloud technique. Prior to the study, users were informed of the anonymous and confidential use of their data and their right to quit the tests at any time. In the first step, a lesson of 20 minutes introduced the principles of editing a MicroApp and the main features of the prototype. Analogously, 20 minutes more were devoted to App Inventor. To give participants more confidence with the two tools, some examples (not related to the tasks, to avoid biasing the experiment) were also presented. The training sessions of the controlled experiment were concluded presenting detailed instructions on the tasks.

After each task and for a given method, participants had to fill in a usability questionnaire. In particular, to measure the usability of the different mobile application generation methods, the Computer Systems Usability Questionnaire (CSUQ) was used [27]. The adoption of a standard questionnaire excludes threats such as poor question wording. CSUQ adopts a 7-point Likert scale. In CSUQ, four subscales provide detailed information on usability aspects of the system. In particular, the subscales are: the overall satisfaction score (OVERALL), system usefulness (SYSUSE), information quality (INFOQUAL) and interface quality (INTERQUAL). After each task, participants also filled in the Post-Task Questionnaire, aiming at assessing whether the laboratory tasks were

TABLE 2: Descriptive statistics and results.

Var.	M_{MA} median	M_{MA} mean	M_{MA} st. dev.	M_{AI} median	M_{AI} mean	M_{AI} st. dev.	p-value	effect size d
Time *	7	7.125	2.37	11	12	4.94	<0.01	medium (-0.74)
Error *	0	0.65	0.95	1	1.32	1.38	0.02	small (-0.37)

* a parametric analysis has been performed through Mann-Whitney test

clear and whether the provided material was enough to perform the activity.

Regarding the preparation of the devices involved in the controlled experiments, we installed a prototype supporting the composition of MicroApps on an Android based Samsung Galaxy S4 device, SDK version 4.2.2, to support the tasks performed using the M_{MA} method, while the App Inventor editor has been installed on a LAN Internet connected PC. The same Android device connected to the PC via USB cable was used in the App Inventor tasks.

The participants accomplished each laboratory session without time limit. For replication purposes, the experiment material is available online[2].

4 Results

In this section, we present the results of the empirical analysis and draw some conclusions with respect to the experiment questionnaires.

- *Objective evaluation.* No participant abandoned the experiment. We summarize the results in the box plots shown in Fig. 12, where the *Time* values concerning the two treatments M_{MA} and M_{AI} are shown On the left-hand side. The results related to the *Error* dependent variable are shown on the right-hand side of the same figure. Table 2 reports descriptive statistics of the dependent variables, grouped by treatment (i.e., M_{MA} and M_{AI}).

Let us note that both the variables *Time* and *Error* have a significant statistical difference with respect to the adopted method, p-value < 0.01 in both cases. To evaluate the magnitude of the effectiveness achieved with the different methods we adopted the *Cohen d* effect size. The effect size is considered negligible for $d < 0.2$, small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$. As shown in Table 2, the effect size is medium and negative for *Time*, and small and negative for *Error*. In addition, 60% (resp. 45%) of participants generated a correct application with M_{MA} (resp. M_{AI}). These results show that MicroApp has a little advantage with respect to App Inventor when considering errors, while the advantage grows in case of the accomplishment time, as better detailed by the box plots in Fig. 12. These results are positive, also considering the availability of a keyboard and a larger screen when using App Inventor with respect to those available on a mobile device.

- *Subjective evaluation.* Concerning the system usefulness and the quality of the information provided,

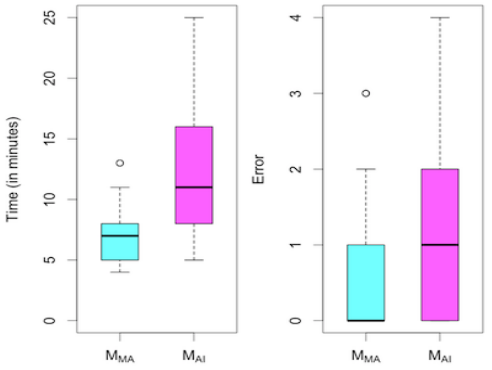


Fig. 12: BoxPlots of the empirical analysis.

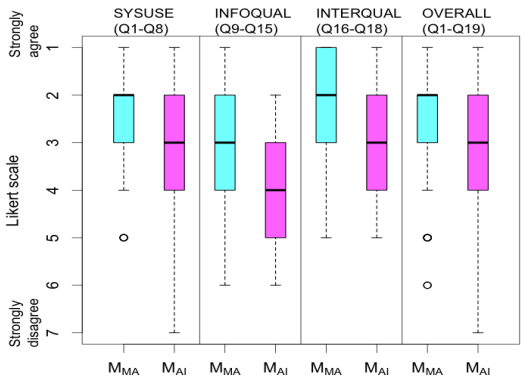


Fig. 13: BoxPlots of the user perception questionnaire.

the participants diffusely had a positive perception of both the systems. As shown in the box plots in Fig. 13, MicroApp Generator is perceived better than App Inventor. Perceptions decreased in the case of the interface quality (INFOQUAL) for both the systems. Also in this case, MicroApp Generator reached better results. In addition, the results of the Post-Task questionnaire revealed that the objective was clear for both tasks and methods. The activity to be performed was perceived as a bit clearer in M_{MA} for task T_1. The provided material was positively evaluated in all cases, while some additional difficulties has been perceived during M_{AI} sessions with respect to M_{MA} ones.

5 Related Work

Many tools for mobile end-user development and (pervasive) service composition are available. In order to allow an effective comparison with respect to MicroApp Generator, in Table 3 we classify the features of each tool depending on the following characteristics.

2. www.unisa.it/docenti/ritafrancese/ricerca/microappgenerator

TABLE 3: Technological features of related works and tools.

Tool	Primitive Components	Language	Expressive Power	Target Users	Development Device	Target Device	Cloud Services
MobiOne	NC	TBT		D	PC	Sm	Y
ContextStudio	NC	TB	Pre	EU	Sm	Sm	N
MobiDev	NC	Vis		D	Sm	Sm	N
App Inventor	SC, NC	Vis	C,L	EU	PC	Sm	Y
Cabana	-	Vis, Txt	C,L	D	PC	Sm	N
Puzzle	SC, SDC, NC	Vis		EU	Sm	Sm	Y
TouchDevelop	SC, NC	TBT	C,L	D	Sm	Sm	Y
Microservice	SC	TBT	C,L	D	Sm	Sm	N
HUSKY	SC	Txt	C	EU	PC	PC	N
Marmite	SC	TB	C	EU	PC	PC	N
ICrafter	SC	TB		EU	PC	PC	N
iCAP	SC	TB	Pre	EU	PC	Sm	N
Pocket Code	NC	Vis, Txt	C,L	EU	Sm	Sm	N
MicroApp [11] [12]	NC	Vis	IL	EU	Sm	Sm	N
MicroApp Generator	SC, SDC, NC	Vis	Pre, IL	EU	Sm	Sm	Y

- *Primitive Components.* The kind of primitive components that the tool can use to compose applications are classified as follow: *Service Components* (SC) can use web and Internet services; *Sensor and Domotics Components* (SDC) can handle sensor data and networks; *Native Components* (NC) can use 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 & Textual* (TBT), only simple apps can be programmed by using the template-based metaphor, whilst the others need textual programming; *Textual* (Txt) programming.
- *Expressive Power.* The tool uses the following constructs for programming the application logic: *Condition* (C), *Pre-condition* (Pre), *Loop* (L), *Indirect loop* (IL).
- *Target Users.* It specifies if the tool is *end-user* (EU) oriented (i.e., no programming skills are required); *Developer* (D) oriented (i.e., programming skills required).
- *Development Device.* It identifies the minimum hardware required to run the development environment: *Smartphone* (Sm); *Personal Computer* (PC) to locally generate the mobile application; *Server* (Se) to remotely generate the mobile application.
- *Target Device.* The final execution device on which the generated application will run: *Smartphone*; *Personal Computer*.
- *Cloud services.* The environment requires the access to cloud services for generating the app.

MobiOne [21] is a Windows-based tool for creating cross-platform mobile applications which are stored in a cloud repository [21]. The tool creates the GUI of the application by means of a device emulator on a PC. Then it connects to an App Center Builder on a remote server to generate the mobile application, which is then downloaded on the PC. For simple predefined tasks, such as *Go to Url*, *Go Back*, *Send SMS*, there is no need of writing code, but more specific actions need to be defined through JavaScript.

The tool ContextStudio [23] [26] supports the user in the definition of context-action rules (triggers) aiming at activating mobile phone functions when the rule conditions are satisfied. The tool, running on a smartphone, provides a selection of contextual triggers and application actions. The triggers can include implicit (context) inputs, such as location, noise and device activity, or explicit input actions, such as gestures or RFID-based commands. The user can include more than one trigger in the rule, while a single rule can contain only one action. The tool interface enables the user to specify the name of the rule and select the action and the triggers among the ones available in the corresponding list. It does not support service composition, but only the triggering of native functionalities of the device.

MobiDev is a system that allows building mobile interfaces directly on a mobile device [35]. In addition, the authors provide functionalities to transform UI sketches created on paper into a mobile User Interface. The definition of the application logic is partially supported by creating the User Interface storyboard and connecting the GUI sketches with arrows. Code entry is needed for processing the user inputs and for refining the application logic.

MIT App Inventor has been described in Section 3. It uses a visual programming language supporting all the programming constructs and a server to store projects and generate .apk files. However, there are stand-alone versions that run entirely on the PC.

Cabana is a web-based application supporting the development of multiple mobile platforms [19]. Programming is based on a wiring diagram supplemented by the use of JavaScript. It is addressed to beginner computer science students.

Danado and Paternò [14] [15] adopt a jigsaw approach to compose pervasive/web services. They use colors for specifying data types, following the idea presented in [12]. The tool Puzzle needs the support of an external server to manage external objects and the application repository. It also provides an authoring tool for designing the User Interface, an HTML viewer and native modules to exploit device functionalities. Puzzle does not support static parameter definition, thus, it does not exploit the advantage of having a predefined application, with some information bounded at design time. Their approach gets the list of applications from an external Application Repository, without contingency management.

Microsoft TouchDevelop [39] is a programming environment running on smartphones. The user writes scripts by tapping on the screen. It has built-in primitives which make it easy to access the sensor data available on a mobile device. It supports the combinations of phone sensor data (e.g., location) and the cloud (via services, storage, computing, and social networks). Differently from MicroApp Generator, the language is not graphical: it uses variables and assignment statements. The smartphone needs to be connected to the TouchDevelop server in order to

generate an app.

In [13] a mobile tool has been proposed to compose Microservices. They can be created considering two user expertise levels: beginner, enabling a template-based development of a Microservice, and advanced, based on an XML-based language. Experienced users may use a visual editor for editing the XML describing Microservices' profile, content, logic, and presentation [16].

The HUSKY tool [36] enables the PC-users to compose the logic of a PC-application by spatially arranging the component services within spreadsheet cells, following the idea presented in [24]. The execution proceeds on the spreadsheet from left to right and from top to bottom. A set of adjacent cells makes a sequence of events. The information regarding the flow of data among cells is not graphically represented.

Marmite [41] is an end-user mashup composition tool. The system runs on the PC. It offers a set of operators such as *Search*, *Extract* and *Filter by*, to extract and process data from web pages and web services. A data-flow approach is adopted to chain operators. The flow of data is displayed in a table, adopting a spreadsheet view.

ICrafter [32] supports PC users in on-the-fly aggregation of services and interface generation using patterns of interfaces.

The iCAP system [18] allows PC end-users to visually design context-aware mobile applications based on if-then rules, temporal and spatial relationships. Parameter types are explicitly declared as in textual programming language.

Pocket Code [8] uses a block-based visual language to create mobile apps on the smartphone. It enables users to create applications exploiting objects. Each object consists of costumes (images), sounds, and actions. Actions are built with lego-like bricks and let the user specify variables, logical operators, loops, and conditions, and access to device sensors.

In the initial idea [11] [12] , MicroApps were graphically composed using only Native Components. Program constructs were represented using rounded rectangles with colored input/output ports that constrain the way in which pieces can be connected. The app components were implemented as native modules on the device and colors were adopted for specifying data types [12]. No external service was accessed.

The comparison summarized in Table 3 highlights that MicroApp Generator is more oriented to end-users than other development environments supporting service composition.

6 Final Remarks

This paper presents MicroApp Generator, an end-user environment which supports the generation of pervasive mobile applications, MicroApps, directly on the smartphone. MicroApps are graphically developed by composing native device features with web services, domotics and sensor management services.

We also evaluated the MicroApp Generator usability by comparing its use with respect to the well known MIT App Inventor. The results of this investigation provide evidence that, even if the mobile interface is restricted in size, the users took less time and made fewer errors.

Future work will be devoted to enable the users to tweak the GUI to match their needs and to port MicroApp Generator on other Mobile Operating Systems. We will also add adaptive controls that allow the generated apps to handle the different capabilities provided by devices, such as the camera resolution.

At present, the MicroApp Generator has been implemented as an Android app. Since the MicroApp-Store shares the MicroApp design, a MicroApp written on one platform will be portable on all other platforms running MicroApp Generator.

The analysis of the state-of-the-art concerning end-user pervasive service composition directly on the smartphones revealed that a tool satisfying these requirements like MicroApp Generator was lacking. In fact, it addresses a new mobile computing scenario where the end-users actively access and compose their personal information, Internet of Things and services available on the web, to create new services. The approach used in MicroApp Generator enriches the smartphone features with capabilities for interacting with remote systems and sensors. It goes towards a new technological trend where smartphones will replace the Personal Computers thanks to their native possibility of wide connectivity, localization and context awareness.

References

[1] *KNX Specification, Version 1.1, Konnex Association, Diegem*, 2004.
[2] App Inventor, *MIT Center for Mobile Learning*. http://appinventor.mit.edu/explore, 2013.
[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Comp. Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
[4] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
[5] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Softw. Eng.*, vol. 10, no. 6, pp. 728–738, 1984.
[6] J. Brnsted, K. Hansen, and M. Ingstrup, "Service Composition Issues in Pervasive Computing," *IEEE Pervasive Computing*, vol. 9, no. 1, pp. 62–70, 2010.
[7] M. M. Burnett, *Visual Progr.* John Wiley & Sons, Inc., 2001.
[8] Catrobat, *Pocket Code*. http://www.catrobat.org, 2014.
[9] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web Services Description Language (WSDL) 1.1," 2001.
[10] P. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in *IEEE Workshop on Vis. Languages*, 1989, pp. 150–156.
[11] S. Cuccurullo, R. Francese, M. Risi, and G. Tortora, "A Visual Approach supporting the Development of MicroApps on Mobile Phones," in *Intl. Conf. on Distributed Multimedia Systems (DMS)*, 2011, pp. 171–176.
[12] ——, "MicroApps Development on Mobile Phones," in *Intl. Symp. on End-User Development (IS-EUD)*, 2011, pp. 289–294.

[13] J. Danado, M. Davies, P. Ricca, and A. Fensel, "An authoring tool for user generated mobile services," in *Conf. on Future Internet*, 2010, pp. 118–127.
[14] J. Danado and F. Paternò, "A prototype for EUD in touch-based mobile devices," in *IEEE Symp. on Vis. Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 83–86.
[15] ——, "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones," in *HCSE*, 2012, pp. 199–216.
[16] M. Davies, F. Carrez, D. Urdiales, A. Fensel, M. Narganes, and J. Danado, "Defining User-generated Services in a Semantically-enabled Mobile Platform," in *Intl. Conf. on Inf. Integration and Web-based App. & Services (iiWAS)*. ACM, 2010, pp. 333–340.
[17] A. De Lucia, R. Francese, M. Risi, and G. Tortora, "Generating applications directly on the mobile device: an empirical evaluation," in *Intl. Conf. on Adv. Vis. Interfaces (AVI)*, 2012, pp. 640–647.
[18] A. K. Dey, T. Sohn, S. Streng, and J. Kodama, "iCAP: Interactive prototyping of context-aware applications," in *Pervasive*, 2006, pp. 254–271.
[19] P. E. Dickson, "Cabana: a cross-platform mobile development system," in *SIGCSE*, 2012, pp. 529–534.
[20] European Commission, *Report of the workshops on the Common Strategic Framework for Research and Innovation: Inclusive, Innovative and Secure Societies Challenge*. http://ec.europa.eu/research/horizon2020/pdf/workshops/inclusive_innovative_and_secure_societies_challenge/summary_report_workshops_on_27_june_2011_and_13_july_2011.pdf, 2011.
[21] Genuitec, *MobiOne*. http://www.genuitec.com/mobile.
[22] T. Green and M. Petre", "Usability analysis of visual programming environments: A cognitive dimensions framework," *Vis. Languages and Computing*, vol. 7, no. 2, pp. 131–174, 1996.
[23] J. Häkkilä, P. Korpipää, S. Ronkainen, and U. Tuomela, "Interaction and End-User Programming with a Context-Aware Mobile Application," in *Intl. Conf. on HCI (INTERACT)*, 2005, pp. 927–937.
[24] D. D. Hoang, H.-y. Paik, and B. Benatallah, "An analysis of spreadsheet-based services mashup," in *Australasian Conf. on Database Tech. (ADC)*, 2010, pp. 141–150.
[25] G. W. Johnson, *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group, 1997.
[26] P. Korpipää, E.-J. Malm, T. Rantakokko, V. Kyllonen, J. Kela, J. Mantyjarvi, J. Häkkilä, and I. Kansala, "Customizing User Interaction in Smart Phones," *IEEE Pervasive Comp.*, vol. 5, no. 3, pp. 82–90, 2006.
[27] J. R. Lewis, "IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use," *Hum.-Comput. Interact.*, vol. 7, no. 1, pp. 57–78, 1995.
[28] N. Mehandjiev, A. Namoune, U. Wajid, L. Macaulay, and A. Sutcliffe, "End User Service Composition: Perceptions and Requirements," in *IEEE European Conf. on Web Services (ECOWS)*, 2010, pp. 139–146.
[29] A. Namoun, T. Nestler, and A. De Angeli, "End User Requirements for the Composable Web," in *Workshop on HCI and Services (HCI)*, 2009.
[30] ——, "Conceptual and usability issues in the composable web of software services," in *Intl. Conf. on Current trends in web eng. (ICWE)*. Springer-Verlag, 2010, pp. 396–407.
[31] T. Nestler, M. Feldmann, A. Preuner, and E. Schill, "Service Composition at the Presentation Layer using Web Service Annotations," in *Intl. Conf. ComposableWeb*, 2009, pp. 63–68.
[32] S. R. Ponnekanti, B. Lee, A. Fox, O. Fox, T. Winograd, and P. Hanrahan, "ICrafter : A Service Framework for Ubiquitous Computing Environments," in *Ubicomp*. Springer-Verlag, 2001, pp. 56–75.
[33] A. Repenning and A. Ioannidou, "What Makes End-User Development Tick? 13 Design Guidelines," in *End User Development*, ser. Human-Computer Interaction. Springer Netherlands, 2006, vol. 9, pp. 51–85.
[34] R. V. Roque, "OpenBlocks : an extendable framework for graphical block programming systems," in *Master Thesis Massachusetts Institute of Tech.*, 2007.
[35] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt, "Mobidev: a tool for creating apps on mobile phones," in *Intl. Conf. on HCI with Mobile Dev. and Serv. (MobileHCI)*. ACM, 2011, pp. 109–112.
[36] D. Skrobo, *HUSKY: A Spreadsheet for End-User Service Composition*. http://www.fer.unizg.hr/_download/repository/DanielSkrobo_KvalifDrIspit_HUSKY.pdf, 2011.
[37] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: an empirical study," *Commun. ACM*, vol. 26, no. 11, pp. 853–860, 1983.
[38] S. Thne, R. Depke, and G. Engels, "Process-Oriented, Flexible Composition of Web Services with UML," 2002.
[39] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich, "TouchDevelop: programming cloud-connected mobile devices via touchscreen," in *SIGPLAN Symp. on New ideas, new paradigms, and reflections on prog. and softw. (ONWARD)*. ACM, 2011, pp. 49–60.
[40] J. Wilcox, *Gartner: 185B mobile app downloads by 2014*. http://betanews.com/2011/01/26/gartner-185b-mobile-app-downloads-by-2014, 2014.
[41] J. Wong and J. I. Hong, "Making mashups with marmite: towards end-user programming for the web," in *SIGCHI Conf. on Human factors in computing systems (CHI)*. ACM, 2007, pp. 1435–1444.

Rita Francese is Assistant Professor at the University of Salerno since 2004. She is co-author of more than 70 papers published in scientific journals or proceedings of refereed conferences. Her research interests concern software engineering, empirical evaluation, human-computer interaction, collaborative work and learning, e-learning, visual languages, and mobile application development.

Michele Risi received his University degree in Computer Science in 2001 and his Ph.D. degree in Computer Science from the University of Salerno, Italy, in 2005. His research interests include grammar formalisms and parsing techniques for visual languages, sketch understanding, architecture and design pattern recovery, reverse engineering of web applications, human-computer interaction, empirical evaluation, data-warehouse and data visualization, and mobile development and applications.

Genoveffa Tortora is a full professor in Computer Science at the University of Salerno, since 1990, where she has been Department Chair, and then Dean of the Faculty of Sciences. She has co-authored more than 250 papers published in scientific journals or proceedings of refereed conferences, and has co-edited three books. Her research interests are in the software engineering and information systems areas, and include human-computer interaction, visual languages, databases, data-warehouses, geographic information systems, image processing and biometric systems.

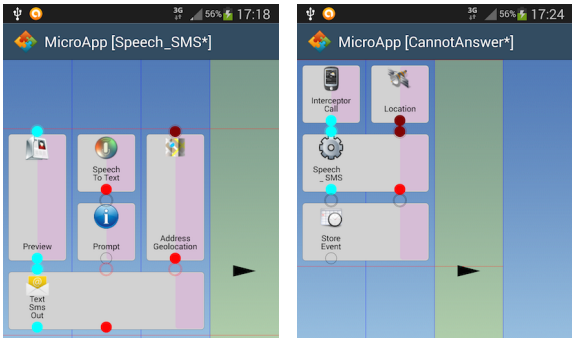
Maurizio Tucci received the Laurea degree in Computer Science from the University of Salerno, Italy, in 1988. He is a full professor of Computer Science at the University of Salerno, since 2000. He was the director of the Department of Mathematics and Computer Science from 2000 to 2006. His research interests include formal models and tools for visual environment design and their applications to visual systems development, image indexing techniques providing an efficient mean to content-based retrieval of images and software engineering.

APPENDIX

EXAMPLE A. Rita is a painter. She cannot answer phone calls during her activity. Thus, she would like to send an SMS possibly by dictating it to the smartphone, sending it, and then saving her location, the caller contact and the SMS text message. She finds on the MicroAppStore the MicroApp *Speech_SMS* (see Fig. 14(a)), which inputs a contact and the GPS position; then it requires the vocal message and sends an SMS with the corresponding text message to the specified contact. At the end, this MicroApp outputs the contact and the text message.

Rita decides to solve her problem by composing a MicroApp *CannotAnswer* by using Speech_SMS. She downloads it into the Service Repository from the MicroAppStore; thus it can be used similarly to all the other services. The parameters of Speech_SMS are automatically determined from its DAG.

Figure 14(b) shows the solution to Rita's problem.

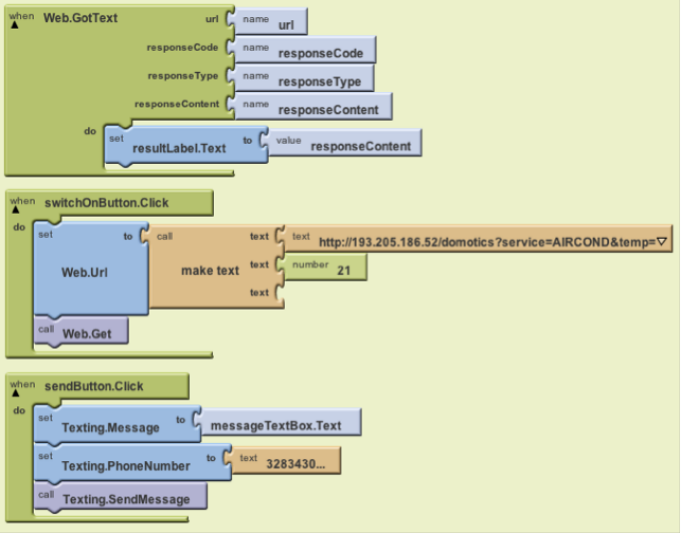


(a) (b)

Fig. 14: The MicroApp service Speech_SMS (a) and its reuse (b).

EXAMPLE B. Figure 15 shows the App Inventor solution of the *"I'm at home"* application, described in Section 2.

(a)



(b)

Fig. 15: The user interface (a) and the visual composition (b) of the application *"I'm at home"* in App Inventor.