Developing Smart Agriculture Applications: Experiences and Lessons Learnt

Isaac Nyabisa Oteyo, Kennedy Kambona, Jesse Zaman, Wolfgang De Meuter, and Elisa Gonzalez Boix
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussels, Belgium

***Abstract*—Smart agriculture applications are promising to improve traditional agriculture in developing regions. Farmers use these applications to collect data on agricultural activities and monitor different conditions in modern farms. However, from the development point of view, these applications are often implemented using text-based programming languages that require experienced programmers. Visual languages can be an alternative to text-based languages that may allow non-expert programmers (*e.g.,* farmers) to implement these applications and make adjustments to existing ones. This study considers two exemplars of concrete and deployable smart agriculture applications (*WebLog* and *UtafitiLog*) developed using two different technologies. The study defines a set of parameters to evaluate these exemplars from which lessons are drawn that can be generalised to other smart agriculture applications.**

***Index Terms*—mobile applications, smart agriculture, distributed computing**

I. Introduction

Smart agriculture aims to improve processes in modern farms to handle the increasing global demand for food. In developing regions, smart agriculture is partially realised by means of distributed mobile applications running on smartphones. Farmers in these regions have the applications installed and configured on their smartphones from where they can collect data and monitor environmental conditions on the farm. Farming activities in these regions happen in small scale farms [1] typically located in remote areas that are faced with intermittent network connection issues [2]–[5]. These intermittent network connection issues contribute towards lowering adoption rates for smart agriculture applications in developing regions. Also, the nature of modern farms requires distributed mobile applications that can be reconfigured to serve different crops and farming seasons [6] *i.e.,* the data parameters to be collected vary for various crops and farming seasons. These applications require to be extended and reconfigured to meet the varying data parameters for various crops or farming seasons. Hence, developing mobile software applications is emerging as one of the vital sectors in smart agriculture to promote sustainable food security [7]. However, programmers face several challenges while developing these applications [8]; *(i)* changing application scenarios and as a consequence application requirements, *(ii)* limited development time, and *(iii)* intermittent network connection issues. During implementation, developers can use either text-based or visual languages to address the aforementioned concerns and the resulting challenges.

This study considers exemplars of two concrete and deployable smart agriculture applications (*WebLog* and *UtafitiLog*) implemented using two different technologies. The study defines a set of parameters to evaluate these exemplars from which lessons are drawn that can bee generalised to other smart agriculture applications. This study is guided by the following research question;

- *How do text-based and visual languages compare when used to implement smart agriculture applications?*

In the subsequent sections, the paper is organised as follows. Section II presents the motivation and running example on smart agriculture applications. This is followed by the application implementations in section III. Section IV presents the evaluation and comparison, while section V presents the discussions and the lessons learnt from the implementation. Section VI presents the related work and lastly, section VII presents the conclusions and gives directions for future work.

II. Smart agriculture application

In this work, we focus on small-scale farmers in developing regions requiring distributed mobile smart agriculture applications to collect different kinds of data. The data is collected in a remote farm that is faced with intermittent network connection issues. Data gathered from a variety of weather sensors in the farm (*e.g.,* temperature and humidity sensors) is uploaded automatically to a server and can be used by the farmer to make informed decisions on required agronomic practices such as initiating programmed misting of crops when temperatures exceed a set threshold. Changes in the data collected from the weather sensors are expected to reflect in real-time once uploaded onto the server. Using the distributed mobile application, the farmer can scan a crop label to invoke an interactive data input form that is adjustable depending on parameters of interest. These parameters may vary with regard to the farming season or type of crop, among others.

To guide our implementation, we derived the following functional requirements; *(i)* user login to access application services, *(ii)* viewing weather data on a dashboard, *(iii)* tagging and scanning plant labels, *(iv)* generating data collection surveys, *(v)* saving collected data to a database, *(vi)* fetching and receiving data from weather sensors, *(vii)* generating alerts based on set thresholds, and *(viii)* storing data on the mobile device when the network becomes unavailable.

From this application example, we consider the following (non-functional) requirements that form the key features for modern smart agriculture applications in developing regions.

- *Offline accessibility:* The application should continue functioning whenever the network connection becomes unavailable. The application should use a client-side database to store data locally on the mobile device while waiting for the network connection to be regained.
- *Reactivity:* The application should respond to events originating from the external environment in real-time. Also, the application should support generating relevant notifications as soon as data from sensors is received to motivate farming decisions, *e.g.,* misting crops when temperatures exceed certain thresholds.
- *Reconfigurability:* It should be possible to change and adapt existing application components to fit different scenarios after the initial development. For example, it should be possible to change the data collection survey to meet data collection requirements for different farming seasons or crops.
- *Extensibility:* It should be possible to add new features and services to the application after implementation, *e.g.,* by supporting Applications Programming Interfaces (APIs) or reusable components.

Based on these requirements, our goal was to get insights on developing smart agriculture applications for small-scale farmers in developing regions. We conducted preliminary experiments to compare the technologies used in implementing the exemplar applications and provide the lessons learnt. The implemented applications meet the requirements derived in this section. *WebLog* was implemented using React-Native framework for JavaScript while *UtafitiLog* was implemented using *DisCoPar* [9]. *DisCoPar* is a reactive visual domain-specific language (VDSL) specially designed for implementing citizen science observatories in which applications are represented as visual flow-graphs [10].

III. Application implementations

In this section, we describe the implementation details for *WebLog* and *UtafitiLog* following the requirements identified in section II. These requirements were translated into application modules to support modular design with APIs. The development process followed an incremental and iterative mechanism. This enabled documenting the design process, results, and lessons learnt throughout the development phases. One developer was engaged in implementing both applications. As previously mentioned, *WebLog* and *UtafitiLog* were implemented using text-based and visual languages respectively.

A. Application Architecture

The architecture for both applications is presented in Fig. 1. Both applications provide similar services that are linked to a Node.js server. The applications consist of an Android mobile application to gather data, a server to store the collected data and generate notifications, and a web client that acts as a dashboard to the server. Both applications are targeted to run on the Android platform since it is a popular platform in developing regions [11], [12]. As shown in Fig. 1, the mobile application and web dashboard communicate with the server via RESTful APIs, HTTP requests, and web sockets. Sensor data coming from the sensor cloud is saved to the server from where it is reactively pushed to the mobile client and dashboard using web sockets. Communication between the server and clients happens through web sockets.

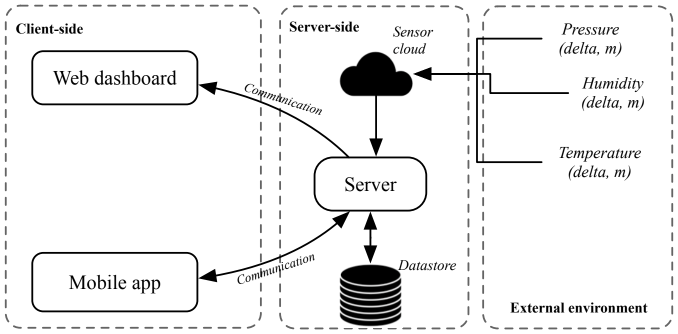

Fig. 1: High level distributed application architecture

B. WebLog

In *WebLog*, the mobile application, server, and web dashboard are implemented in JavaScript. React-Native and ReactJS frameworks were used to implement the mobile application and web dashboard respectively. The server was implemented using Node.js linked to MongoDB for data storage. The mobile application is composed of different parts (views/screens) implemented as React-Native components (*e.g., Login*, *Scan*, *SensorData*, *DataCollection* components etc.) to realise the services the application offers. The *application dashboard* is the entry point that anchors all application views to one central navigation point. From the anchor point, users can navigate to the various sections of the application.

WebLog adheres to the previously identified requirements as follows;

1) Offline accessibility: WebLog uses a client-side database that runs on the mobile device to store data whenever the network becomes unavailable. The application keeps a counter that indicates to the user the number of unsynchronised records. If a farmer is collecting plant morphological data and a network disconnection occurs before sending the data to the server, the data is saved locally awaiting the network reconnection. When the network becomes unavailable, the application creates a database on the mobile device to save data as illustrated in Fig. 2. The application uploads locally stored data to the remote server when the network becomes available and deletes the local copies to free up storage space.

2) Reactivity: Code Listing 1 shows the specification for sending weather data to clients via sockets. Clients have to register to receive updates from the defined socket. New data updates are automatically pushed to the client (line 14 in code Listing 1). For example, the application provides a service to

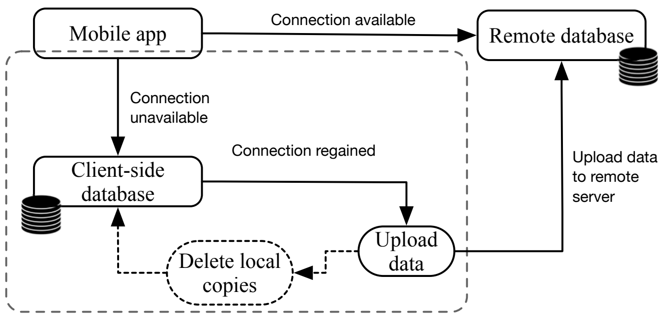

Fig. 2: Offline accessibility in *WebLog*

monitor temperature, humidity, and pressure conditions. This data is displayed on a component that is updated whenever the server receives new data.

```
sendSensorDataToClients = data => {
   const socketIds = Object.keys(this.
      sockets);
   let allSockets = this.sockets;
   if (socketIds.length === 0) {
     console.log("No client sockets.");
     return;
   }
   const dataToSend = {
       Temperature: data.temperature,
       Humidity: data.humidity
     };
   console.log("Sending sensor data!");
   socketIds.forEach(socketId => {
       allSockets[socketId].emit("
          socket_name", dataToSend);
   });
};
```

Listing 1: Sending weather data to clients via sockets

Also, notifications are generated based on set thresholds. For example, to generate alerts, we can set the threshold values at 25°C and 35% for temperature and humidity in tropical regions respectively.

3) Reconfigurability: *WebLog* can be used to collect different kinds of data. Each time a data collection instance is created, it requires generating the corresponding data collection survey. Also, threshold values to generate notifications in *[R_2]* can be changed to fit different scenarios. The data collection survey and threshold values for notifications can be changed by non-experienced programmers *e.g.,* farmers via provided reconfiguration interfaces.

4) Extensibility: *WebLog* has been implemented as a textual component-based system with APIs that allow extending the application with new services. However, this extensibility is done in code since *WebLog* was implemented using a textual language.

C. UtafitiLog

As mentioned before, *UtafitiLog* was implemented using *DisCoPar*, a VDSL for visual programming designed for building applications for citizen science observatories (CSOs) [10]. In *DisCoPar*, programs are constructed by means of visual components. These components have an execution scope *i.e.,* mobile, server, and web. To build an application, the components are wired together into a flow-graph using arcs and application components use these arcs to exchange data. Fig. 3 shows the application flow-graph for *UtafitiLog*. The flow-graph is read from left to right. Each connection has an arrow on it indicating the direction of data flow. By design, the application flow-graph is a directed acyclic graph. Each component is coloured based on its execution scope[1]. This execution scope facilitates building distributed applications.

Components have `output` ports for emitting data and `input` ports for receiving data. These ports are typed based on the data that they emit or receive. On the application flow-graph (Fig. 3), each port is colour-coded with the type of data it emits or receives[2]. The arcs connecting components on the application flow-graph assume the colour of the origin port. These arcs represent data dependencies and always flow from the output ports of source components to the input ports of destination components. To explain these data dependencies, we split them into output and input data dependencies. Each component has at least one input or output data dependency with some components having multiple data dependencies because of having multiple input or output ports.

To implement *UtafitiLog*, we first identified the components that we required. The application flow-graph illustrated in Fig. 3 uses 26 components. Out of the 26, we reused 20 components from the existing component library. We implemented the remaining 6 components and published them to the component library. In regard to component scope, 6 components execute on the mobile device (*i.e., Scan*, *ObservationPopSurvey*, *BufferObservation*, and *Label*), 9 components execute on the web (*i.e., Label, LineChart, Alert, and Table*), and 11 components execute on the server (*i.e., SensorWeather*, *Rounding*, *IsGreaterThan*, *AddDataToObservation*, *ObservationDatabase*, and *ObservationToTable*).

In Fig. 3, *SensorWeather* receives temperature, humidity, and pressure weather data from the sensor cloud and passes it as `numeric` values to the *Rounding* components that are used to set the precision level. *SensorWeather* component emits `numeric` data values which are accepted as input to the *Rounding* component. Data values (`numeric`) from the *Rounding* component are passed to the *Label* components for display and the *LineChart* components for visualisation on the dashboard. To generate notifications, the data is passed

[1]*Black* coloured components execute on mobile device; *grey* coloured components execute on the web; and *light-grey* coloured components execute on the server.

[2]*Blue* coloured ports receive or emit `numeric` data values; *black* ports receive or emit `any` data values; *orange* coloured ports emit or receive `boolean` values; *green* coloured ports emit or receive `observations`; and *yellow* coloured ports emit or receive `datasets`

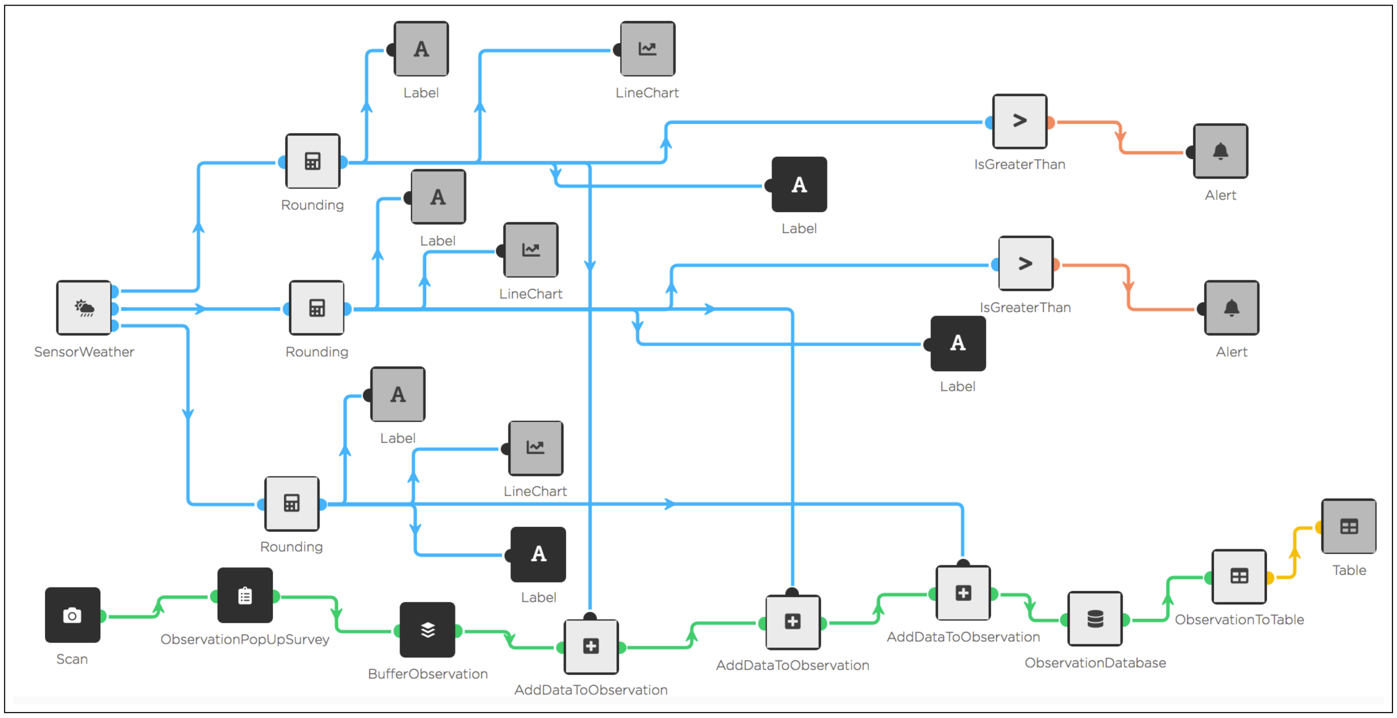

Fig. 3: *UtafitiLog* flow-graph showing component connections from left to right. The arrows on the connections show the direction of data flow. *Black*, *grey*, and *light-grey* coloured components execute on the mobile, web, and server in that order.

through *IsGreaterThan* component where thresholds are set. *IsGreaterThan* component accepts `numeric` data values and outputs `boolean` values. The input value is compared to the set threshold and if it evaluates to `true`, a notification is fired. The *Alert* component is used to display the notification message on the dashboard. Scanning of plant labels is done via the *Scan* component while creating data collection surveys is done using the *ObservationPopUpSurvey*. *BufferObservation* component supports offline accessibility while *Observation-Database* component connects the application to the database. *AddDataToObservation* component is used to aggregate data (observations) from different components.

UtafitiLog adheres to the previously identified requirements as follows;

1) Offline accessibility: The *BufferObservation* component is used to support offline accessibility by keeping copies of data on a client-side database each time the network breaks. The data collected when the network is unavailable is sent to the remote server when the network is regained.

2) Reactivity: *UtafitiLog* exploits the reactive and event-driven design of *DisCoPar* VDSL ensuring the application components continue executing whenever they receive data on their input ports. The execution of one component in the application flow-graph triggers the execution of subsequent components that depend on the output from the previous component. Also, notifications are generated based on threshold values. In our application, using *IsGreaterThan* component we can set 25°C and 35% threshold values for temperature and humidity in a tropical region respectively.

3) Reconfigurability: The data collection survey is implemented as a reconfigurable component to allow collection of different kinds of data. For example, each time the farming season changes and the data collection parameters change, the survey component is reconfigured to reflect the new data parameters. Also, threshold values for generating notifications can be changed for different farming scenarios. The *ObservationPopUpSurvey* and *IsGreaterThan* components are easily reconfigurable with minimal programming experience *e.g.,* by farmers.

4) Extensibility: Adding new features or functionalities to the application is done by incorporating components from the component library to the application flow-graph. The application flow-graph can also be updated by removing components. Both adding and removing components can be done even when the application is already deployed.

D. Application Services

To provide the functionality described in section II, both applications provide similar services *i.e., (i)* user registration, *(ii)* plant label scanning, *(iii)* generating data collection surveys, *(iv)* generating alerts based on weather information, and *(v)* visualising weather information on a dashboard. These applications present two perspectives to users; one perspective for logged in users and another for visitors. By default, a landing page is presented to a user visiting the application and is not yet logged in. This default landing page allows the user to navigate to the registration page to create an account or login page for authorisation and authentication to proceed to other application services. User registration is a public service that allows users to sign up into the application. Once logged in, the user can proceed to the application dashboard from where one can navigate to different sections of the application. The scanning service uses the device camera to read the plant labels and invoke the data collection surveys which are generated in both applications. Also, the alerts in the two applications are displayed on the web dashboard based on adjustable thresholds.

IV. Evaluation and Comparison

In this section, we compare *WebLog* and *UtafitiLog*. We use the features derived in section II as the primary criteria for qualitative evaluation. Since both applications support similar features and functionalities, and both are implemented in the same software stack, we postulate that both implementations exhibits similar performance. Therefore, the evaluation in this study is focused on the following questions;

- *[EQ$_1$] Which strategies are followed to implement the requirements identified in section II?*
- *[EQ$_2$] What is the disk space utilisation for the implemented applications?*
- *[EQ$_3$] How do the implemented applications compare to other smart agriculture applications?*

A. Implementation Strategies

In this section, we compare the implementation strategies used by both applications to adhere to the requirements in section II.

1) Offline accessibility: Both applications support offline accessibility by keeping copies of data in a client-side database whenever the network becomes unavailable. Different strategies exist in the literature for implementing offline accessibility *e.g.,* caching, local storage, and client-side database [13]–[15]. Caching and local storage have limitations of size which are addressed by the client-side database strategy [16]. The use of a database running on mobile devices for data storage significantly eliminates the need for infrastructure to host database servers in the farm. It also eliminates the need for middleware to cache data in the cloud.

2) Reactivity: Both applications support reactivity in different ways. *UtafitiLog* exploits the reactive and event-driven design of *DisCoPar* VDSL that ensures application components continue executing as long as they keep receiving data on their input ports. Execution of one component in the application flow-graph triggers the execution of subsequent components that depend on the output from the previous component. *WebLog* utilises reactive libraries to implement web sockets that push data to clients. Clients have to register to receive data from these sockets; new data is automatically pushed to the client.

3) Reconfigurability: Both applications support reconfigurability to adapt to different requirements within the farm. For example, changing the data collection surveys to fit different crops or farming seasons, the two applications make use of reconfigurable data collection surveys. This makes it easier for the farmer to reuse the same applications to collect different kinds of information on the farm for different farming seasons or various crops.

4) Extensibility: Both applications are extensible to add new features and functionalities. Since each service in *WebLog* is implemented as a textual component, extending the application requires coding. Extensibility in *UtafitiLog* happens at two levels; *(i)* at the code level to add new components to the component library and *(ii)* the application flow-graph level to add or remove features from an existing application. Also, both applications support APIs for programmers to extend application services. Since both applications are implemented as components (either textual or visual components) they support variability. In this case, variability allows reconfiguring and extending the applications to meet new or changing requirements *i.e.,* adaptation [17]. For example, anticipating changing thresholds to generate alerts and allowing the user to change them appropriately. The user can also specify the alert messages and change the data collection surveys. Application flow-graph for *UtafitiLog* is changeable to add or remove components. Changes done at the application flow-graph level require minimal programming experience and therefore can be considered suitable for non-experienced programmers.

B. Disk Space Utilisation

Farmers in developing regions operate in remote areas and in most cases depend on paid mobile data to download and install applications on their mobile devices. The cost of mobile data positively correlates to the size of applications to download. Therefore, in this section, we measured *(i)* the size of the compiled application *i.e.,* downloadable APK size, and *(ii)* disk space utilised by the installed application on the mobile device. The second parameter is important especially for resource constrained devices that, typically, have limited persistent memory available. In terms of downloadable APK file size, *WebLog* and *UtafitiLog* resulted in 25 MB and 10.8 MB respectively. In terms of disk space, *WebLog* utilised 122 MB, while *UtafitiLog* used 36.04 MB. The substantially more disk space used by *WebLog* could be as a result of insufficient compaction of the application bytecode. Also, React Native builds native libraries for `armebi` and `x86` device architectures into the same APK, hence the larger APK size in *WebLog* compared to *UtafitiLog*. To minimise this size, it requires creating APKs for each device architecture.

C. Comparison to other Smart Agriculture Applications

The applications presented in Table I support different activities on the farm *e.g.,* crop disease diagnosis [18]; monitoring [19]–[23]; crop nutrient computation [24]; data collection [13], [25]; and cost management [26]. Only MobiCrop [13] supports offline accessibility through caching which has limitations of storage size. Unlike most of the existing smart agriculture applications, *WebLog* and *UtafitiLog* are offline accessible, reactive, and reconfigurable. The implementations support APIs and components that allow extending the applications to add new services.

TABLE I: Comparing smart agriculture applications

	Offline accessible	Reactive	Reconfigurable	Extensible
MobiCrop [13]	✓*	✗	✗	✓†
Blynk [21]	✗	✗	✓*	✗
AgDataBox [25]	✗	✗	✗	✓†
SmartHof [22]	✗	✗	✗	✓†
Connected Farm [23]	✗	✗	✗	✓†
CLUeFARM [27]	✗	✗	✓*	✓†
WebLog	✓**	✓	✓**	✓†
UtafitiLog	✓**	✓	✓**	✓††

✓* caching; ✓** client-side database; ✓* widgets; ✓** configurable surveys; ✓† REST APIs; ✓†† REST APIs and visual components

V. Discussion and lessons learnt

In this section, we discuss how text-based and visual languages compare in implementing smart agriculture applications. We re-examine the overall research question and provide the lessons learnt from the implementation.

A. Text-based vs. Visual Languages

The overall research question in this paper is *how text-based and visual languages compare when used to implement smart agriculture applications*. We address this research question with the specific research evaluation questions [EQ_1], [EQ_2], and [EQ_3] introduced and described in section IV.

Different visual programming languages already exist in the literature. In our implementation, we limit ourselves to *DisCoPar* VDSL which falls within the family of flow-based visual programming languages. In general, these languages require modelling software applications as directed graphs that connect networks of nodes. The nodes exchange data via arcs connecting them. Applications take different names depending on the language *e.g.,* `flows` (Node-RED[3], FRED [28], and D-NR [29]), `pipes` (NoFlo[4], WotKit [30]), and `process` (MsgFlo[5]). Also, these languages provide different *constructs* for application development *e.g.,* `nodes` (Node-RED, D-NR, FRED, and NoFlo), `components` (MsgFlo, *DisCoPar*), and `processors` (WotKit). The languages further provide *channels* to link different logical entities of an application. These *channels* are mainly `wires` (Node-RED, D-NR, and FRED) and `edges` (NoFlo, MsgFlo, and WotKit). Applications in these languages are presented as graphs and application components react to any incoming data on their input ports. Apart from *DisCoPar*, none of the other surveyed languages supports mobile applications *i.e.,* they do not support exporting application flow-graphs for execution in mobile platforms.

Writing software using textual languages still dominates the current practice in application development [31]. In this paper, our choice for React-Native framework for *WebLog* implementation was motivated by its support for cross-platform development. Similarly, *DisCoPar* VDSL supports cross-platform development in addition to the visual components. Though textual languages have been used in software development for years they still face some limitations that can be overcome by visual languages, *e.g.,* the cognitive load that they impose on non-expert programmers as opposed to visual languages [32]. Even though, visual languages promise a better alternative to application development, the visual components are still implemented using textual languages. This means the underlying architectures for visual languages have to be properly and adequately crafted to allow easy addition of new visual programming constructs. Both text-based and visual languages can be used to develop smart agriculture applications with similar features. However, we take the view that text-based languages require adequate prior programming experience compared to visual languages. From the implementation experience, several differences become apparent and we raise them in the subsequent section as lessons learnt.

[3]https://nodered.org
[4]http://noflojs.org
[5]https://msgflo.org/

B. Lessons Learnt

Even though the approaches used in implementing our application yield two concrete and deployable applications with the same functionalities, there are key lessons learnt from the implementation and these are;

1) Programming effort and code reusability: Every functionality in *WebLog* was implemented from scratch and this required more programming effort. On the contrary, components were reused to implement *UtafitiLog*; only a few additional components were programmed to meet the application implementation requirements. *UtafitiLog* took faster to yield an application compared to *WebLog*, but required some changes to be done on *DisCoPar*. In general, visual components developed to implement *UtafitiLog* were reusable and helped reduce on the overall time taken to implement the application.

2) Cross-platform development: Though our applications were targeted for the Android platform, our implementation technologies support cross-platform development. This gave us room to write non-platform specific code that could be compiled to respective target platforms. This leads us to think that cross-platform development can serve different farmers with varied mobile hardware and software configurations.

3) Programming experience: Overall, while implementing *WebLog* required more programming expertise, using components to implement *UtafitiLog* required less programming expertise. Since, *UtafitiLog* demonstrates the simplified expressive power of VDSLs in representing software applications, we believe non-experienced programmers like farmers can use such technology to reconfigure their smart agriculture applications.

4) Software development support: Relying on libraries and tools helped in implementing the application requirements and evaluating the performance of the implemented applications. As such, using software tools like editor, debuggers, monitors, and profilers boosts the process of implementing and evaluating smart agriculture applications.

5) Distributed applications: Both approaches support implementing distributed applications. However, *UtafitiLog* uses scoped components that make implementing distributed applications easier for non-experienced programmers compared to *WebLog*. In *WebLog*, the client and server-side were implemented separately before linking them together.

6) Performance impact: There is no significant performance impact for the technologies used in implementing the two applications. Hence, we argue in support of the notion that using either a textual or a visual language yields applications with similar performance. This further implies that visual languages can be used as an alternative for building smart agriculture applications.

VI. Related work

In this section, we present the related work on studies documenting experiences in implementing smart agriculture solutions and applications. To the best of our knowledge, there is no published report documenting experiences on implementing mobile applications for smart agriculture. We therefore, first, base our related work on experience reports for smart agriculture solutions addressing some of the issues described in section II. Secondly, we present related work on textual vs graphical representation of software applications.

A. Smart Agriculture Solutions

Several studies document experiences in implementing Internet of Things solutions for smart agriculture [33]–[35]. For instance, Gunasekera *et al.* [33] document their experiences in implementing IoT solutions to meet different changing user requirements with less programming effort and also offer offline accessibility. Though the work promises to offer offline accessibility, this is done by providing a feature to export data into CSV files for offline accessibility. Jayaraman *et al.* [34] report on building a smart agriculture platform that can allow adding new sensors with ease and allow real-time analysis of data coming from the sensors. The study supports "do-it-yourself" concept for non-experienced programmers to add new sensors to the platform and extend its functionality. In their implementation, they use query-processing to fetch data and trigger events.

B. Textual vs Graphical Software Representation

Several studies have focused on textual and graphical representation of software. For instance, Heijstek *et al.* [36] analyse the effectiveness of visual or textual artefacts in communicating software design decisions. The findings show that neither diagrams nor textual descriptions is significantly more efficient to communicate software design decisions to developers. However, this may not apply to non-experienced developers as highlighted in the study done by Jolak *et al.,* [37] (preprint). The findings in this study show that describing software designs graphically is better than textual descriptions *i.e.,* graphical descriptions promote better recall for developers.

Labunets *et al.* [38] investigate comprehending software risk models using graphical, tabular, and textual notations. The findings show that the tabular notation is comprehensible in both recall and precision. However, this is still subject to cognitive complexity of software tasks.

Other studies have focused on making programming more available to naive developers. For instance, the study done by Mason and Dave [39] compares block-based (visual) vs flow-based programming for naive developers. With visual programming, non-programmers do not need to have the same investment in particular syntax or semantics for textual languages.

Lastly, Sharafi *et al.* [40] investigate modelling and presenting software requirements using graphical vs. textual representations. The findings show no significant difference in using either textual or graphical representations. However, the study notes that training developers can significantly improve the efficiency of using graphical representations.

VII. Conclusion and future work

In this paper, we sought to understand how text-based and visual languages compare in implementing smart agriculture applications. We implemented exemplar smart agriculture applications using two different technologies resulting in two concrete and deployable applications with similar functionalities *i.e., WebLog* and *UtafitiLog*. *WebLog* was implemented using a text-based programming language while *UtafitiLog* was implemented using a VDSL. The findings show that both languages can be used to develop smart agriculture applications with similar performance. However, text-based languages require adequate prior programming experience compared to visual languages. VDSLs may provide a better alternative to allow non-experienced programmers (*e.g.,* farmers) to adapt applications for different uses with minimal prior programming experience. Future work entails conducting controlled experiments to determine the cognitive load that textual and visual languages have on experienced and non-experienced application developers.

Acknowledgment

This work is supported by the Legumes Centre for Food and Nutrition Security (LCEFoNS) programme which is funded by VLIR-UOS. The programme is a North-South Collaboration between the Katholieke Universiteit Leuven, Vrije Universiteit Brussel (both in Belgium) and Jomo Kenyatta University of Agriculture and Technology (Kenya).

References

[1] H. B. Fiehn, L. Schiebel, A. F. Avila, B. Miller, and A. Mickelson, “Smart agriculture system based on deep learning,” in *Proceedings of the 2nd International Conference on Smart Digital Environment*. New York, NY, USA: ACM, 2018, pp. 158–165.

[2] M. Bacco, P. Barsocchi, E. Ferro, A. Gotta, and M. Ruggeri, “The Digitisation of Agriculture: A Survey of Research Activities on Smart Farming,” *Array*, vol. 3-4, pp. 1–11, 2019.

[3] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, and E.-H. M. Aggoune, “Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk,” *IEEE Access*, vol. 7, pp. 129 551–129 583, 2019.

[4] M. O’Grady, D. Langton, and G. O’Hare, “Edge computing: A tractable model for smart agriculture?” *Artificial Intelligence in Agriculture*, vol. 3, pp. 42–51, 2019.

[5] A. Eitzinger, J. Cock, K. Atzmanstorfer, C. R. Binder, P. Läderach, O. Bonilla-Findji, M. Bartling, C. Mwongera, L. Zurita, and A. Jarvis, "GeoFarmer: A monitoring and feedback system for agricultural development projects," *Computers and Electronics in Agriculture*, vol. 158, pp. 109–121, 2019.

[6] A. Kamilaris and A. Pitsillides, "Mobile Phone Computing and the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 885–898, 2016.

[7] C. S. M. Babou, B. O. Sane, I. Diane, and I. Niang, "Home edge computing architecture for smart and sustainable agriculture and breeding," in *ACM International Conference Proceeding Series*, 2019, pp. 1–7.

[8] S. Al-Ratrout, O. Husain Tarawneh, M. HusniAltarawneh, and M. Yosef Altarawneh, "Mobile Application Development Methodologies Adopted in Omani Market: A Comparative Study," *International Journal of Software Engineering & Applications*, vol. 10, no. 2, pp. 13–22, 2019.

[9] J. Zaman, K. Kambona, and W. De Meuter, "DISCOPAR: A Visual Reactive Programming Language for Generating Cloud-Based Participatory Sensing Platforms," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. New York, NY, USA: ACM, 2018, pp. 31–40.

[10] J. Zaman and W. De Meuter, "DisCoPar: Distributed Components for Participatory Campaigning," in *2015 IEEE International Conference on Pervasive Computing and Communication Workshops*, 2015, pp. 160–165.

[11] K. Heimerl, A. Menon, S. Hasan, K. Ali, E. Brewer, and T. Parikh, "Analysis of Smartphone Adoption and Usage in a Rural Community Cellular Network," in *ACM International Conference Proceeding Series*, vol. 15, Singapore, 2015, pp. 1–4.

[12] S. Ahmad, A. L. Haamid, Z. A. Qazi, Z. Zhou, T. Benson, and I. A. Qazi, "A view from the other side: Understanding mobile phone characteristics in the developing world," in *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, Santa Monica, CA, 2016, pp. 319–325.

[13] R. K. Lomotey, Y. Chai, A. K. Ahmed, and R. Deters, "Distributed mobile application for crop farmers," in *MEDES'13 Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, Luxembourg, 2013, pp. 135–139.

[14] C. Liu, B. C. Fruin, and H. Samet, "SAC: Semantic Adaptive Caching for Spatial Mobile Applications," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA: ACM, 2013, pp. 174–183.

[15] F. A. Marco, J. Gallud, V. M. R. Penichet, and M. Winckler, "A Model-Based Approach for Supporting Offline Interaction with Web Sites Resilient to Interruptions," in *Current Trends in Web Engineering*, Q. Z. Sheng and J. Kjeldskov, Eds. Cham: Springer International Publishing, 2013, pp. 156–171.

[16] A. Wharry, "HTML5 Offline Technologies," 2014. [Online]. Available: http://andreawharry.com/assets/img/portfolio/WharryAndreaFinalPaper.pdf

[17] A. Fortier, G. Rossi, S. E. Gordillo, and C. Challiol, "Dealing with variability in context-aware mobile software," *The Journal of Systems and Software*, vol. 83, no. 6, pp. 915–936, 2010.

[18] M. A. Carmona, F. J. Sautua, O. Pérez-Hernández, and J. I. Mandolesi, "AgroDecisor EFC: First Android™ app decision support tool for timing fungicide applications for management of late-season soybean diseases," *Computers and Electronics in Agriculture*, vol. 144, pp. 310–313, 2018.

[19] M. F. Işik, Y. Sönmez, C. Yilmaz, V. Özdemir, and E. N. Yilmaz, "Precision Irrigation System (PIS) using sensor network technology integrated with IOS/Android Application," *Applied Sciences (Switzerland)*, vol. 7, no. 891, pp. 1–14, 2017.

[20] M. Caria, J. Schudrowitz, A. Jukan, and N. Kemper, "Smart farm computing systems for animal welfare monitoring," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics*, 2017, pp. 152–157.

[21] P. Serikul, N. Nakpong, and N. Nakjuatong, "Smart Farm Monitoring via the Blynk IoT Platform," in *2018 Sixteenth International Conference on ICT and Knowledge Engineering*. Bangkok, Thailand: IEEE, 2018, pp. 70–75.

[22] F. Carpio, A. Jukan, A. I. M. Sanchez, N. Amla, N. Kemper, A. Isabel, M. Sanchez, N. Amla, and N. Kemper, "Beyond Production Indicators: A Novel Smart Farming Application and System for Animal Welfare," in *Proceedings of the Fourth International Conference on Animal-Computer Interaction*, 2017, pp. 7:1–7:11.

[23] M. Ryu, J. Yun, T. Miao, I. Y. Ahn, S. C. Choi, and J. Kim, "Design and implementation of a connected farm for smart farming system," *2015 IEEE Sensors*, pp. 1–4, 2015.

[24] M. V. Bueno-Delgado, J. M. Molina-Martínez, R. Correoso-Campillo, and P. Pavón-Mariño, "Ecofert: An Android application for the optimization of fertilizer cost in fertigation," *Computers and Electronics in Agriculture*, vol. 121, pp. 32–42, 2016.

[25] C. L. Bazzi, E. P. Jasse, P. S. Graziano Magalhães, G. K. Michelon, E. G. de Souza, K. Schenatto, and R. Sobjak, "AgDataBox API – Integration of data and software in precision agriculture," *SoftwareX*, vol. 10, pp. 1–8, 2019.

[26] Y. Murakami, "iFarm: Development of web-based system of cultivation and cost management for agriculture," in *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems*, 2014, pp. 624–627.

[27] M. Colezea, G. Musat, F. Pop, C. Negru, A. Dumitrascu, and M. Mocanu, "CLUeFARM: Integrated web-service platform for smart farms," *Computers and Electronics in Agriculture*, vol. 154, pp. 134–154, 2018.

[28] M. Blackstock and R. Lea, "FRED: A Hosted Data Flow Platform for the IoT," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, 2016, pp. 2:1–2:5.

[29] N. K. Giang, M. Blackstock, R. Lea, and V. C. Leung, "Developing IoT applications in the Fog: A Distributed Dataflow approach," in *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015*. IEEE, 2015, pp. 155–162.

[30] M. Blackstock and R. Lea, "IoT mashups with the WoTKit," in *Proceedings of 2012 International Conference on the Internet of Things, IOT 2012*. Wuxi, China: IEEE, 2012, pp. 159–166.

[31] J. A. Wang, "Towards component-based software engineering," *Journal of Computing in Small Colleges*, vol. 16, pp. 182–194, 2000.

[32] K. J. Harms, "Applying cognitive load theory to generate effective programming tutorials," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, 2013, pp. 179–180.

[33] K. Gunasekera, A. N. Borrero, F. Vasuian, and K. P. Bryceson, "Experiences in building an IoT infrastructure for agriculture education," *Procedia Computer Science*, vol. 135, pp. 155–162, 2018.

[34] P. P. Jayaraman, A. Yavari, D. Georgakopoulos, A. Morshed, and A. Zaslavsky, "Internet of things platform for smart farming: Experiences and lessons learnt," *Sensors*, vol. 16, no. 11, pp. 1–17, 2016.

[35] N. Dlodlo and J. Kalezhi, "The internet of things in agriculture for sustainable rural development," in *Proceedings of 2015 International Conference on Emerging Trends in Networks and Computer Communications*, 2015, pp. 13–18.

[36] W. Heijstek, T. Kühne, and M. R. V. Chaudron, "Experimental analysis of textual and graphical representations for software architecture design," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 167–176.

[37] R. Jolak, M. Savary-Lelanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gerard, and M. Chaudron, "Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication," *EMSE Journal*.

[38] K. Labunets, F. Massacci, and A. Tedeschi, "Graphical vs. tabular notations for risk models: On the role of textual labels and complexity," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2017, pp. 267–276.

[39] D. Mason and K. Dave, "Block-based versus flow-based programming for naive programmers," in *2017 IEEE Blocks and Beyond Workshop*. IEEE, 2017, pp. 25–28.

[40] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y. Guéhéneuc, "An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension," in *2013 21st International Conference on Program Comprehension*. IEEE, 2013, pp. 33–42.