An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud

Timo Aaltonen*, Varvana Myllärniemi†, Mikko Raatikainen†, Niko Mäkitalo* and Jari Pääkkö†
*Department of Pervasive Computing
Tampere University of Technology, Tampere, Finland
Email: {timoaaltonen, niko.makitalo}@ut.fi
†School of Science, Department of Computer Science and Engineering
Aalto University, Espoo, Finland
Email: {varvana.myllarniemi, mikko.raatikainen, jari.paakko}@aalto.fi

Abstract—Nowadays, smart devices have excellent computing power and connectivity and at the same time are used for various purposes. This creates the possibility for complex, cooperative multi-device programs. However, current programming paradigms are not tailored for such a setting. This paper tackles the problem by introducing a new paradigm: an action-oriented programming model. Actions are proactively and pervasively initiated pieces of functionality, which provide synchronized and coordinated joint behavior between several devices. We show how the action-oriented programming model can be realized with a device cloud infrastructure. As a concrete example of an action-oriented infrastructure, we discuss the Social Devices Platform that demonstrates a socio-digital system between devices and people in proximity.

I. INTRODUCTION

In recent years, smart devices have become increasingly capable and connected. They are used for everyday purposes: for entertainment, for socializing with friends, and for sharing life events. Continuous connectivity enables the devices to perform tasks at the background and even be controlled by other devices. However, currently available applications typically operate on one device at a time. This calls for a new breed of multi-device applications that enable coordinated interaction between devices in a pervasive manner. Such coordinated interaction between devices can be realized with mobile cloud computing, which in its simplest form refers to accessing cloud computing resources from a mobile device [1], but often also the ability to share services among devices in the same cloud is assumed [2]. In a mobile cloud, the devices can be any devices with the necessary capabilities, such as mobile phones, smart TVs, laptops, and even heart rate monitors.

Consider the following example. When Alice meets her friends in the cafeteria, her phone recognizes that friends are nearby and suggests that Alice could share and present the photo album from her latest trip. If Alice agrees, the photos are browsed through collectively, and each photo is shown on the friends’ screens at the same time. This example involves coordinated and synchronized functionality among a number of devices to provide user-visible behavior; we call such functionality an action. Actions can start proactively and pervasively, even without user initiation, if the given preconditions are met. For example, the photo sharing action is initiated when Alice has a new photo album and approaches her friends. Nevertheless, a user is always in control such as being able to abort an action.

However, supporting such a scenario requires additional effort with traditional programming models. This is because the programming constructs do not directly support necessary primitives, such as actions and triggers. Thus, we introduce an action-oriented programming model to enable easy and efficient development of coordinated, proactively and pervasively initiated, multi-device programs that use actions as their basic building blocks. We also discuss the characteristics of the mobile cloud infrastructure that needs to realize the action-oriented programming model. As a concrete example of such an infrastructure, we discuss the Social Devices Platform (SDP) [3], which is a cloud-based infrastructure for enabling co-located interaction between multiple users and devices. We use a PhotoSharing action from the SDP as a running example throughout the paper.

Compared to our earlier work [3], this paper contributes the following. Firstly, we establish and define the action-oriented programming model and its concepts; these are applicable to any application domain. Secondly, we describe the actors, artifacts and services of the infrastructure that is needed to realize an action-oriented programming model infrastructure.

The rest of the paper is organized as follows. The running example is explained in Section II. We introduce the action-oriented programming model in Section III, and explain how the SDP realizes the programming model in Section IV. Section V outlines the related work, while the results are discussed in Section VI. Finally, Section VII concludes.

II. THE RUNNING EXAMPLE: PHOTO SHARING

The PhotoSharing action is used as a running example to explain the motivation and the basic concepts of the action-oriented programming model.

In the running example, Alice meets Bob, Carol, and Dan at a cafeteria. When joining the table, her phone recognizes that her friends are nearby and suggests that Alice could share and present the photo album from her latest trip. Although Alice has already previously added her travel photos to Flickr, the cafeteria provides a socially convenient moment for telling about the trip. Instead of squeezing in to see the photos on Alice’s screen, or everyone individually browsing the photos,
the friends are able to see the photos on their own devices in a synchronized manner, and Alice can comment on each photo.

More specifically in the PhotoSharing action, when Alice approaches her friends, her phone recognizes and notifies her of the possibility of sharing the new photos. This notification may utilize several forms of interaction, such as talking aloud, vibration, and UI screens. After Alice indicates that she wants to share the photos, Bob and Carol agree to join in the session with the UI that appeared on their devices. Since Dan’s phone is running out of battery, he does not join. Alice browses her photos and selects the photos to be viewed. At the same time, the photos appear in Bob’s and Carol’s devices, while Alice tells about her trip. Additionally, Alice has recorded audio related to some photos, which is played by Bob’s device as it has the best loudspeaker.

The PhotoSharing action highlights key characteristics and requirements of the action-oriented programming model. All functionality is built around actions: an action provides proactively initiated, typically user-visible, coordinated behavior, such as photo browsing, in a synchronized manner between multiple devices. To participate in an action, each device must have installed the required capabilities. Actions can start proactively, even without user assistance, but a user is able to control whether the action continues. Therefore, actions can define triggers, upon receipt of which the action is tried to be executed. In addition, the necessary preconditions must be satisfied before the action can take place.

III. THE ACTION-OrientED PROGRAMMING MODEL

In this section, we describe the action-oriented programming model in more depth. A program in the action-oriented programming model consists of one or more actions, one of which executed by an initiating trigger. We exemplify the programming model and its concepts via the PhotoSharing running example. The action definition for the running example, written in the Python programming language, is presented in Figure 1. However, the concepts of action-oriented programming (see Table I) are not specific to any programming language. The PhotoSharing example presented in Section II illustrates several key concepts of the action-oriented programming model (see Table I).

Firstly, an action developer must be able to easily define actions. Within an action, the developer should be able to easily define coordinated behavior among several devices (in the example, collective photo viewing). To control whether and when actions are executed, the action developer can attach preconditions (e.g., being close enough to friends and having enough battery), and potential triggers (e.g., new photos) to actions. While actions define behavior, the device capabilities (e.g., a capability to view a picture) provide the functionality. The capabilities are defined in interface. When writing actions, an action developer can reuse existing, or expect development of new, capabilities. These issues are relevant for all action-oriented programs and, thus, are summarized in Table I and explained in more detail in the rest of this section.

Secondly, various dynamic information may need to be managed, e.g., to evaluate the preconditions and recognize triggers. In PhotoSharing, the infrastructure that implements action-oriented programming must be able to observe Alice’s Flickr account, friendship relations, battery levels, and the proximity of Alice’s friends. However, the characteristics of the dynamic information are specific to the infrastructure that implements the action-oriented programming model. For example, the social aspect of the SDP necessitates that the proximity of the devices must be known to perform an action. The dynamic information in the exemplary SDP infrastructure is explained in more detail in Section IV.

A. Devices and Capabilities

Since actions involve coordinated interaction between devices, the devices must possess capabilities that can be invoked during action execution. A capability means that the device is able to carry out the task associated with the capability, and there is a means to use the capability proactively. For example, to be able to collectively view photos, Alice’s friends have all installed the Screen capability to their devices, and Alice has installed the ImageBrowser capability.

From the implementation perspective, the capabilities of devices are manifested as interfaces. An interface consists of operations; for example, the ImageBrowser interface contains an operation browse. Thus, an interface is a contract about the functionality that a device fulfills. Although the action-oriented programming model is not specific to any programming language, the interfaces can be implemented, e.g., as objects of Python classes. In order to be able to utilize the capabilities in actions, the devices must register the interfaces they implement. Thus, the action-oriented programming model assumes a device registry that contains information about devices and their capabilities. However, the registry can be centralized or decentralized, long-term or ad hoc, depending on how the programming model has been realized.

The programming model itself does not set any specific requirements for device capabilities or the number of interfaces that a device should implement. However, a device with no capabilities cannot actively participate in any action, since an action can only invoke devices through their interfaces. Further, capabilities are reusable across several actions: the developer may even write new actions without implementing any device-specific code. This avoids the potentially costly implementation of device-specific functionality. For example,
a developer can reuse the image browsing capability in any action by declaring that a device in an action role r should implement the ImageBrowser interface. Thereafter, all devices with the ImageBrowser capability installed can be assigned to r. To prevent any issues of concurrent operation invocations, a device can participate in only one action execution at a time.

B. Actions

An action is the basic unit of modularity, encapsulating the joint behavior of several devices, and defining roles, parameters, preconditions, and a body.

First, an action defines roles and parameters. Before action is executed, devices are assigned to roles (in which they participate in the action). For example, a device whose owner is a “source” of the photos is assigned to the role src in the running example. Parameters, on the other hand, are values that are passed to an action when it is executed; for example, the albumURI parameter refers to an album of images.

Second, an action defines a boolean-valued expression precondition, which must evaluate to true for the participating device and parameter combination. In the example, the precondition ensures that the owners of the devices are friends on Facebook and have the required capabilities.

Third, an action defines a body: the execution of the body results in action-specific behavior. The body defines how an action is orchestrated by describing how operations are invoked for the devices assigned to the roles. For example, the notification of an action being started can be invoked by src.notification.notify(message, ['Yes', 'No'])), since role src has been declared to implement the capability Notification. This invocation causes the device to notify the user in a user-defined visible way (the device may say aloud or display the message depending on the user’s preferences), and the user is shown buttons with labels ‘Yes’ and ‘No’. If she decides to start the action, first the device which has the best audio quality is chosen (Line 15 in Figure 1). The set self.participants contains all participants of the action (in this case src and sinks), and chooseOne is a function, which selects the device that has the best quality related to the capability (‘AudioDevice’) given as a parameter. Then the execution continues so that the device assigned to the src roleBrowse the photos (Line 16) and when the user shares a photo, the photo is displayed on the screens of the devices assigned to the sink roles (Line 23). Some photos have a related audio recording, which was recorded when the photo was taken. This audio is played by the bestAudioDevice if both the device and the audio exist. In general, the body can invoke the operations of devices either sequentially or in parallel. A parallel block is started by calling the startParallelExecution function (Line 21) and ended by calling endParallelExecution.

Finally, in order to be able to initiate and execute actions, the action-oriented programming model assumes an action registry.

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action description</td>
<td>A piece of code that defines the action, its precondition, device roles, and the action body.</td>
</tr>
<tr>
<td>Interface descrip-</td>
<td>A piece of code that defines an interface, which consists of a set of operations.</td>
</tr>
<tr>
<td>Trigger description</td>
<td>A piece of code that defines the trigger: a trigger notifies when the action is tried to be scheduled.</td>
</tr>
<tr>
<td>Device capability</td>
<td>Software resource that implements a certain interface for a given device. By installing a certain device capability to her device, the user indicates her ability to participate in actions that utilize this capability.</td>
</tr>
</tbody>
</table>

C. Triggers and Scheduling

In addition to defining what the action does, the action-oriented programming model provides triggers, which are constructs for initiating an action proactively. When a trigger is received, a related action is attempted to be executed. Although the use of triggers is not necessary in the action-oriented programming model, triggers are convenient for many practical implementations, since a continuous evaluation of the precondition can be too resource-intensive.

The use of triggers requires explicit definition of the triggers and the mapping between triggers and actions. Figure 1 illustrates the definition of PhotoSharingTrigger, which is associated to the PhotoSharing action using the getTriggers method. In general, there can exist a many-to-many relation between triggers and actions. The triggers and the corresponding actions need to be registered in a triggering table, which is hosted by a trigger registry.

Based on an incoming trigger, there needs to be a way to select one action from the set of actions related to a trigger. This process, called scheduling, also involves finding an action, whose precondition is satisfied by a set of devices. In some cases, it may be possible to assign some devices to action roles in advance, whereby the scheduling process can be eased. For instance, the source of a trigger might be assigned to a specific role in an action.

There may also be a need to manage the lifespan of incoming active triggers. For this purpose, all incoming triggers can be stored in an active triggers table to wait for rescheduling if no action preconditions are immediately satisfied. Different strategies can be employed for rescheduling active triggers. For instance, scheduling can be reattempted after a certain amount of time, or after a predefined change in a state. Also, any failed triggers can simply be discarded.

Finally, the action-oriented programming model does not restrict the source of triggers, i.e., a trigger can be sent by any source, such as a dedicated trigger service, or a device.

D. Artifacts and Infrastructure

So far, the action-oriented programming model has been discussed from the concept point of view. For the developers to write action-oriented programs, the concepts presented in Table I need to be encapsulated into artifacts. The necessary programming artifacts for the action-oriented programming model are listed in Table II.

However, in order to realize the action-oriented programming model and to be able to schedule and execute the actions, a supporting infrastructure is needed. Thus, there is a need for the action-oriented programming model infrastructure. There are several services that the supporting infrastructure needs to implement; Table II summarizes the needed services that were identified earlier. That is, there needs to be a registry for actions, devices and triggers, as well as a service for scheduling and execution of actions. An example realization of an action-oriented programming model infrastructure is discussed in the next Section.

Fig. 2 illustrates how different actors interact with the artifacts and the action-oriented programming model infrastructure. Firstly, the infrastructure provider deploys and administers the action-oriented infrastructure. Secondly, the capability developer is responsible for implementing the capabilities to various devices adhering to contract specified in interface descriptions. Thirdly, the action developer writes action descriptions. Since the action developer does not know the identities or characteristics of the devices, she uses interfaces.
to refer to the required devices and their capabilities. Fourthly, the user installs capabilities to her device. Finally, at runtime, the devices register to the infrastructure, and constantly report dynamic information about their state.

However, it must be noted that the order of the events illustrated in Fig. 2 may be slightly different. Typically, the action developer reuses existing device capabilities, since writing code for new device capabilities to potentially several mobile platforms is an arduous task. However, for certain new innovative actions, it may very well be that the action developer has to implement device capabilities from scratch, which would imply that Steps 2 and 3 in Fig. 2 are interleaved.

To summarize, the programmer needs to do the following to create an action-oriented program. First, the programmer needs to reuse, or define and implement interfaces that capture the necessary capabilities of the devices. Second, the programmer needs to define the actions as well as preconditions for starting the action. Finally, the programmer needs to define the triggers that initiate the actions.

IV. IMPLEMENTING THE ACTION-ORIENTED MODEL: CASE SOCIAL DEVICES

In the following, we discuss an example of an action-oriented programming model infrastructure, the SDP. The SDP is built around the concept of Social Devices [3]. Social Devices enrich co-located interactions by augmenting the social interaction of people that meet face-to-face, by making interaction between several devices visible to the user, and by enabling more intuitive, discussion-based interaction between users and devices. The SDP implements all necessary services of the action-oriented programming model: it schedules actions and assigns participants when triggers arrive; orchestrates the action execution; and manages the device and action registries and trigger tables. Additionally, the SDP manages dynamic information of device proximities and states.

A. The SDP Architecture

The architecture of the SDP (Figure 3) consists of clients and a number of cloud services running on the Amazon cloud. The client side currently consists of Android devices and Python capable devices, such as Linux laptops and Meego.
phones. The interactions in Figure 3 exemplify how the photo sharing scenario is executed; Alice and her friends are denoted with A, B, C, and D. At the client side, the device responsibilities include reporting the device capabilities and states that are needed for preconditions, and discovering other devices in their proximity. For the proximity, the devices search periodically for other devices with Bluetooth, measure the signal strength of the found devices and communicate this to the server side. At the server side, ProximityServer is responsible for maintaining the proximity information and notifying about changes in proximity. Therefore, ProximityServer maintains a proximity graph, where nodes represent the devices, and edges include the mutual device distances as an attribute. An edge gets stale and is removed from the graph when a predefined time has passed. This way the graph reflects the current proximity information. The proximity graph can be used for searching for sets of devices in each other’s proximity. In Figure 3, the proximity graph is partitioned into \((B, C, D)\) and \((E, F)\) based on earlier measurements. When Alice gets close to her friends (message #2.2), new edges \((A, B, \text{distance} = 2.2)\) and \((A, D, \text{distance} = 2.7)\) are added to the graph. When such significant changes occur in the proximity graph, ProximityServer notifies Controller (message #3).

Similarly at the server side, StateServer is responsible for managing the device states: the devices report their local states, such as battery and noise level, to StateServer. (The interaction between the devices and StateServer is omitted from Figure 3 for clarity.)

Controller is responsible for managing the device registry, action registry, triggering table, and active triggers table. In addition, it is responsible for the scheduling process, e.g., other services report triggers to it.

To be able to recognize various triggers, the architecture can be integrated with several observer services that are responsible for sending the triggers to Controller. Figure 3 illustrates how FlickrObserver sends PhotoSharingTrigger on behalf of Alice’s device when Alice uploads her photos to Flickr. At this point, Controller attempts to schedule the PhotoSharing action, but since Alice is at home with no nearby friends, scheduling fails. Therefore, Controller stores Alice’s device A and the corresponding active trigger in the ActiveTriggers table, and starts waiting for changes in Alice’s proximity. This is an example of a state-based rescheduling strategy described in Section III-C.

When ProximityServer reports a proximity change in Alice’s device, scheduling can be retried. Controller retrieves the proximity sets and device states, and assigns the device A to the src role in the PhotoSharing action. Thereafter, ProximityServer utilizes CaaS (Configurator-as-a-Service), which is responsible for assigning devices to roles by utilizing an inference engine. Based on the device states, CaaS knows that the battery of D is dry, so only B and C are assigned to the action. CaaS can even make advanced inference in terms of QoS optimization using recommendation techniques [4]. The recommendation techniques take user preferences, device capabilities, and context of the devices into account when making the device selections for particular action roles. Consequently, the best loudspeaker can be selected not only by taking speaker quality into account but also by taking the distance of devices into account. The recommendation might not always result in the best device for each role, but a good enough recommendation can be enough in most situations.

After scheduling and device assignment, Controller sends the action along with the participates to Orchestrator, whose task is to orchestrate the execution of the action body. The orchestration is based on utilizing the device interfaces and their operations (Section III). Since devices are allocated and reserved for one specific action at a time, it is easier for the users to keep track of what their devices are doing. Orchestrator monitors the clients, and in a centralized manner, handles exceptions raised by the devices when the devices fail or stop responding. Invoking the operations of devices is done in a synchronized way, allowing the operation calls to act like any operation calls in Python. For the developers, this allows an illustrative way to coordinate the devices and keep the action body clear. After finishing the body execution, Orchestrator notifies Controller.

B. The Client Architecture

To participate in actions, devices need to have SocialDevicesClient installed and running. The architecture of the client (Figure 4) is modular and based on plugins. Three plugins are essential: OrchestratorPlugin participates in action execution by maintaining a connection to the Orchestrator service and invoking operations according to server coordination; ProximityPlugin measures the signal strength of nearby devices and reports this information along with the Bluetooth MAC identities to ProximityServer, which then translates the information to distance; and StatePlugin reports contextual data and state values, such as battery level or GPS location, to StateServer. In contrast to essential plugins, capabilities and interfaces of the action-oriented programming model are implemented as interface plugins; for example, the ImageBrowser capability is implemented as one plugin. In general, there can be any number of interface plugins.

The whole SocialDevicesClient is implemented as a native application for each environment. While this ensures, e.g., efficiency in terms of battery consumption, separate implementations of SocialDevicesClient and plugins are needed for different environments. However, an interface plugin can be general and used in different actions. The modularity of the client and the thin implementation of the essential plugins
have allowed SocialDevicesClient to be light from the device point of view. Furthermore, the modularity allows users to be in more control over their devices, as they can decide what kind of capabilities their devices have.

V. RELATED WORK

In the action-oriented programming model, the operations and the device coordination are based on actions. The notion of an action is rooted in the DisCo method [5], which is a formal specification method for reactive and distributed systems based on the joint action theory [6]. Unlike DisCo actions that can be mapped to terms of logic, the actions in the action-oriented programming model do not have a formal meaning. In DisCo, the participants are typically processes, whereas in the action-oriented programming model the action participants are, usually, devices. A DisCo action has a guard that is similar to the precondition. The action body in DisCo is an atomic parallel assignment clause, whereas the body in the action-oriented programming model is a normal function.

The execution model of action-oriented programs behaves like a single guarded loop in Dijkstra’s guarded command language (GCL) [7]. In the GCL, the basic building blocks are guarded commands which is a statement list prefixed with boolean expressions. The statement list is eligible for execution only if the boolean expression is true. Similarly, an action can be executed if the precondition is true, and the statement list corresponds to the body of the action. However, since our contribution is a real programming platform, evaluating the preconditions is alleviated by introducing the notion of triggers, whereas the GCL requires continuous evaluation of guards.

While this paper focuses on the action-oriented programming model, the concept of Social Devices is described more thoroughly along with a more detailed technical description in [3]. In particular, the SDP relies on the concepts of knowledge-based product configuration [8] applied to software [9], [10] within the broader research topic of software variability [11] and, in particular, in a form of a dynamic software product line [12].

The previous approaches for multiple device coordination have mainly focused on information presentation techniques (e.g., [13]–[15]), and synchronizing multimedia (e.g., [15], [16]). However, our work is different since we are not aiming to automate services in pre-defined locations, such as smart spaces, but rather to coordinate devices wherever they are in the proximity of each other. Nor are we focusing on generating user interfaces and coordinating them on devices as [13], [14]. Instead, our focus is to make the devices interact and socialize independently, and make the operations visible for the users.

Work has been done on advanced discovery of proximity [17] focusing especially on energy efficient discovery. Furthermore, existing web services such as Facebook places, Foursquare, and Google Latitude take location and proximity into account typically by a means of GPS or network identification. However, these services facilitate actions or interactions between devices and users in proximity at most in a manual and rudimentary manner. In contrast, our focus is on an advanced action-oriented programming model suitable for proximity-based systems whereas the SDP implementation relies on relatively simple Bluetooth discovery of proximity.

The action-oriented programming model can be used to create actions that facilitate the interaction between humans, devices, and between humans and devices. Thus, the actions can be considered a form of groupware [18]. Groupware can be broadly defined as software systems that facilitate the activities and the interaction of a group of people for achieving a common task or goal. Actions differ, however, from traditional real-time groupware applications such as chats or document editors in that they do not necessarily have to be tied to a common task or goal but can be more ad hoc in nature, such as greeting a person.

To contrast our action-oriented programming to pervasive computing [19], [20] or the Internet of Things [21] in general, the actions provide human visible behaviour, instead of performing actions and forming compositions in the background. This is the opposite to pervasive computing that has the vision of indistinguishable form or minimal user distraction [19]. However, the triggering and scheduling of actions takes place in a more pervasive manner. Thus, action-orientedness can be considered as a special class of pervasive computing, where actions carry the user-visible behaviour. Both pervasive computing and action-oriented programming largely rely also on smart devices in an environment.

The action-oriented programming model also has some similarities to service-oriented computing (SOC) [22]. For instance, actions, action bodies, capabilities and interface definitions in the action-oriented programming model correspond roughly to service compositions, workflow processes, basic services and service descriptions in SOC, respectively. Also, the action-oriented programming model is concerned with concepts such as publication of capabilities, action coordination, action composition, and QoS. The action-oriented programming model can, thus, be considered as a practical model of SOC for a pervasive context. However, the action-oriented programming model is more device-centric and involves triggers and proactively initiated service compositions.

VI. DISCUSSION

The action-oriented programming model has and is being assessed in various ways.

From the developer’s point of view, we have evaluated the action-oriented programming model by creating a number of demo applications to the SDP. These applications include Photo Sharing, Car Game (each device is a “camera” to the track, and everybody controls her own car with voice), Greeting Devices (our phones greet each others), and Unread SMS Reminder (the reminder is given by a friend’s device).

With the action-oriented paradigm, development has shown to be fast and intuitive. An undergraduate student was able to create the photo sharing action in a couple of days. Constructing a similar application from scratch would have required concentrating on difficult connectivity and synchronization issues. Now, these are hidden by the concepts of the programming model. To get more feedback from developers, we hired an outside team to design and implement a multiplayer game by using the platform. The application was developed in
cooperation with Demola\textsuperscript{1}, an innovation instrument targeted for fostering innovation and experimenting with radical ideas. Overall the results (reported in [23]) were encouraging as the team was enthusiastic about the concept, and only after short introduction managed to start implementing their application.

While the exemplar infrastructure SDP is in place, the action-oriented programming model needs to be supported with development tools. The current implementation of action-oriented programming model relies on Python with a few specific conventions so that existing Python tools can be utilized helping with syntax checking and highlighting. However, a more specific tools support for an action-oriented programming model could be provided. For example, a development environment by a means of an Eclipse plugin, could help programmer by constructing necessary skeletons and by being integrated with the SP to manage actions and interfaces.

The ongoing CoSMo\textsuperscript{2} research project studies multimodal interaction techniques, and also conducts studies in real context to gain empirical understanding of how users experience proximity based action-oriented applications. Moreover, the project studies how users feel about actions that start proactively, and which kinds of actions are socially acceptable [24]. Yet, further studies on user experience are needed.

Further, the applicability of the action-oriented programming model should be studied beyond the SDP. The programming model seems to be applicable to scenarios that involve coordinated and synchronized behaviour between several distributed entities; the behaviour is then encapsulated into actions that are initiated proactively. A prime example application areas include smart homes, which often involve distributed, coordinated behavior between home appliances and devices as a reaction to a certain condition or user need.

A programming model reflects the system which executes the programs. Successful programming models have a straightforward relation to the system. For example, procedural and object-oriented languages have usually an obvious mapping to the von Neumann architecture on which they are executed. Similarly, the action-oriented programming model as described in this paper reflects the distributed system executing the actions, and, vice versa. This means that future needs arising from new application areas on which the programming model is applied may require changes or additions to the programming model. Similarly, a more formal specification of action-oriented programming model is a future work item.

A future work item for the model includes adding transac-
tionality. An action is a natural unit for transactional execution. In the beginning of the action, devices join the transaction, and in the end, if all went fine, the changes become persistent.

In the programming model and the SDP, action coordination relies on the server side. Some confluence to Message Queue or Publish-subscribe technologies can be seen: Orchestrator could be regarded as the publisher that sends messages to an action-specific message bus. However, unlike in the aforementioned technologies, Orchestrator is always aware of the message receiver and also expects a response, since the operations are invoked synchronously. Moreover, a device is reserved for one action at a time, and hence is subscribed to only one action-specific message bus at a time. However, research on different kinds of coordination approaches is ongoing. For instance, performance overhead could be reduced by one device coordinating the other devices directly with Bluetooth, or even by distributing the coordination among many devices participating in the action.

A more closely related principle is push messaging. The SDP uses a technique called long polling, which is practically a means to implement push messaging in mobile devices. All major mobile phone operating systems have their own similar push messaging services originating from XMPP, such as Google Cloud Messaging, Apple Push Notification Service, and Microsoft Push Notification Service. On the one hand, a similar messaging infrastructure exists in mobile phones and, therefore, the SDP does not require any additional services. Rather, existing services could be utilized. However, the current prototype implementation uses long polling due to its simplicity. On the other hand, the performance or battery overhead caused by utilizing the SDP seems to be low since new services are not required whenever the SDP is inactive. Consequently, the SDP seems to be an efficient implementation.

\section{Conclusions}

We introduced an action-oriented programming model for constructing programs that are executed as actions in a coordinated manner by several devices in pervasive settings. We introduced the key concepts and discussed essential artifacts and services for the programming model.

From the programmer’s viewpoint, the following is needed to create an action-oriented program. First, the programmer needs to reuse, or define and implement interfaces that capture the necessary capabilities of the devices. Second, the programmer needs to define the actions, which describe the joint behaviour of several devices as well as preconditions for starting the action. Finally, the programmer needs to define the triggers that initiate the actions.

To assess the action-oriented programming model, we implemented it in the SDP, which is a platform that aims at enriching co-located interactions between users and devices. Based on the evaluation, the action-oriented programming has enabled efficient and intuitive development of various SDP scenarios. However, the contribution needs to be assessed beyond the technical solution, for example, by studying how users feel about proactively initiated, multi-device actions, and which scenarios are suitable for action-oriented programs.

\section{Acknowledgement}

We acknowledge the financial support of EIT ICT Labs, Academy of Finland (264422), and TEKES as part of the Cloud Software program of DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).
REFERENCES


