MAAT: Mobile Apps As Things in the IoT

WYATT LINDQUIST, Lancaster University, United Kingdom
SUMI HELAL, Lancaster University, United Kingdom
AHMED KHALED, Northeastern Illinois University, United States
GERALD KOTONYA, Lancaster University, United Kingdom
JAEJOON LEE, Lancaster University, United Kingdom

As the Internet of Things (IoT) proliferates, the potential for its opportunistic interaction with traditional mobile apps becomes apparent. We argue that to fully take advantage of this potential, mobile apps must become *things* themselves, and interact in a smart space like their hardware counterparts. We present an extension to our Atlas thing architecture on smartphones, allowing mobile apps to behave as *things* and provide powerful services and functionalities. To this end, we also consider the role of the mobile app developer, and introduce *actionable keywords* (AKWs)—a dynamically programmable description—to enable potential thing to thing interactions. The AKWs empower the mobile app to dynamically react to services provided by other *things*, without being known a priori by the original app developer. In this paper, we present the mobile-apps-as-things (MAAT) concept along with its AKW concept and programming construct. For MAAT to be adopted by developers, changes to the existing development environments (IDE) should remain minimal to stay acceptable and practically usable, thus we also propose an IDE plugin to simplify the addition of this dynamic behavior. We present details of MAAT, along with the implementation of the IDE plugin, and give a detailed benchmarking evaluation to assess the responsiveness of our implementation to impromptu interactions and dynamic app behavioral changes. We also investigate another study, targeting Android developers, which evaluates the acceptability and usability of the MAAT IDE plugin.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → *Architectures*; Embedded and cyber-physical systems.

Additional Key Words and Phrases: Internet of Things, thing architecture, mobile apps, actionable keywords

ACM Reference Format:
Wyatt Lindquist, Sumi Helal, Ahmed Khaled, Gerald Kotonya, and Jaejoon Lee. 2019. MAAT: Mobile Apps As Things in the IoT. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 4, Article 143 (December 2019), 22 pages. https://doi.org/10.1145/3369823

1 INTRODUCTION

The success of the Internet of Things (IoT) will largely depend on how the *things* are architected to opportunistically engage with each other. This is especially the case in personal IoT where the smart cooperation between the services offered by the *things* could expand the ways for smart space users to interact with their smart homes and workplaces. However, such smart engagement cannot be achieved through simple connections between the offered services, but through the dynamic creation of IoT applications and scenarios opportunistically by the *things* themselves.

Authors' addresses: Wyatt Lindquist, w.lindquist@lancaster.ac.uk, Lancaster University, United Kingdom; Sumi Helal, s.helal@lancaster.ac.uk, Lancaster University, United Kingdom; Ahmed Khaled, aekhaled@neiu.edu, Northeastern Illinois University, United States; Gerald Kotonya, g.kotonya@lancaster.ac.uk, Lancaster University, United Kingdom; Jaejoon Lee, j.lee3@lancaster.ac.uk, Lancaster University, United Kingdom.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2019 Copyright held by the owner/author(s).
2474-9567/2019/12-ART143
https://doi.org/10.1145/3369823

At the same time, the opportunistic dynamic development of domain-related applications and scenarios should not only be based on the services offered by the *things* but also on the relationships that could logically and functionally tie these services together. The social networking concepts started to converge with IoT technologies forming a new paradigm named Social Internet of Things (SIoT) [6, 7]. SIoT is about creating a social network of *things* through a set of social relationships and interactions. The recently proposed ideas on social IoT [11] are to logically link the *things* according to their identification attributes (e.g., *things* from same vendor), not on the services offered by these *things*. However, the exploitation of service-level relationships in the context of social IoT adds an effective programming perspective to such a new evolving paradigm. The inter-thing relationships programming framework [17, 18] broadens the social IoT thing-level relationships and utilizes a set of concrete relationships between the offered services to empower a much wider class of meaningful IoT applications.

On the other hand, the typical models of the *things* in smart space are *things* with sensors that sense and collect environment parameters and *things* with actuators that perform actions and change the state of the environment. However, smart spaces are not only full of hardware models of *things* that offer hardware-based services, but also software models [25]. A software model—a mobile app—is a new type of *thing* that represents a different model, offers software-based services and functions, and is able to engage with its mates in the ecosystem in different IoT applications and scenarios.

Consider two *things* that have a potential for interaction—a digital video recorder (DVR) device and a mobile phone "TV guide" app that lists upcoming television programs—but have not been explicitly programmed for each other. The DVR could offer to record the program the user has selected on their phone; however, the developer of the app did not see this as a possibility and did not implement such a feature. This does not stop the *things*, however: they have already exchanged capabilities and created a meaningful interaction based around the concept of TV programs and recording. The user is given the choice (driven by the DVR and displayed by the app) to record the program whose listing they are currently viewing, and to "integrate" such behavior into the app, permanently creating a relationship between the DVR and mobile app. Although the developer did not implement this functionality, such an interaction was able to take place, thanks to the underlying features of the *thing* architecture and the app's descriptive metadata: the developer instead defined keywords—capabilities and interests—that allowed the architecture to suggest new relationships with other *things* in the smart space. Once accepted by the user, the app can adjust its behavior and interface to accommodate new elements for the deduced interaction. In this paper, we utilize the inter-thing relationships programming framework [17, 18] and present an extension, Mobile Apps As Things (MAAT), to our Atlas thing architecture [19] targeting mobile app developers and attempting to pave the way for the mobile apps to engage in smart spaces as *things* in the ecosystem. The extension introduces *actionable keywords* (AKWs) as programmable and dynamic descriptions that enable potential *thing* to *thing* interactions. The AKWs empower the mobile app to opportunistically and dynamically react to functionalities and services provided by other *things* in the ecosystem, without being a priori configured or statically wired by the mobile app developer. For MAAT to be smoothly adopted by developers, we developed an IDE plugin for a common mobile app development environment (Android Studio) to ensure minimal, acceptable, and practically usable experience.

The paper is organized as follows. Section 2 highlights related work and presents a brief summary of the Atlas thing architecture on which we base our ideas in this paper. Section 3 presents the framework by which we propose mobile apps to be redesigned as *things* in an IoT. Section 4 presents the AWK idea as an enabling mechanism for mobile apps to become *things*. Section 5 gives details of our implementation including programming time support in the form of an Android Studio Plugin. Section 6 presents a performance evaluation to test the feasibility of our approach in terms of acceptable responsiveness of mobile apps when they are modified to also be *things*. Section 7 presents an acceptability study with end users (mobile app developers). Finally, a discussion and future work are presented in section 8, and a conclusion is presented in section 9.

2 PRIOR WORK AND REQUISITE BACKGROUND

In this section, we limit our coverage of prior work to research/products that relate most to our idea of a mobile apps as a thing (MAAT). As no prior work exists that attempts to directly affect mobile apps to be things, as with MAAT, we cover mainly ingredients and referential related work.

If This Then That (IFTTT) [13, 29] is a web-based service that allows user in smart space to manually connect the various Internet-based services and features (e.g., Twitter) to develop an application called an applet. IFTTT binds such services and features through a single rule: if a certain event occurs (e.g., received a message) then perform some action (e.g., vibrate the smartphone). Users can give IFTTT permission to utilize various cloud APIs offered by service vendors (e.g., Instagram) to operate on their data through predefined applets, or custom ones they create. IFTTT has also added support for smart products (e.g., Belkin WeMo home automation and Philips Hue lights bulbs), as well as Android system functionality (e.g., Bluetooth, messaging, and notifications). While these features move towards providing *thing*-like behavior on mobile devices, they focus more on cloud-based services and events, and require applications to provide REST-based endpoints for integration. This makes local interactions difficult and requires users to manually specify applets, either custom or community provided.

Similarly, Yun et al. [32] created a prototype named TTEO (Things Talk to Each Other) that can be programmed by user-defined if-then rules, in the same vein as IFTTT but focusing specifically on smart *things*. The architecture consists of two platforms; Mobius, a connectivity platform that resides as an IoT server, and &Cube, is a smart service server that acts as the interaction domain. Mobius communicates with devices, maintains virtual entities for each, and sends this information to &Cube, which allows developers to create and execute new services with registered devices through predefined control statements. This functionality is exposed through a mobile app that allows users to specify these rules on the fly. While the project better handles local *thing*-to-*thing* interactions, it still requires users to think up their own interactions, and does not consider the potential for integration with software features like those in a mobile app.

MOSDEN [26] is an IoT middleware targeting mobile devices, allowing users to collect and analyze sensor data through a service model. Users can connect with new sensor types without the need to directly program such an interaction. The middleware uses a plugin architecture to achieve this; individual plugins (developed by third parties) can be added and removed on the fly to support a specific sensor type or brand, and can be downloaded through the smartphone's application store. The mobile app allows users to view detected sensors and their collected data. While this explores augmenting a mobile app with behavior not explicitly considered by the original developer, it only considers information transmitted to the mobile device, not data or actions that may come *from* the mobile app to be used by other *things*.

Coulson et al. [10] proposed a programming method to facilitate the composition of self-contained systems without relying on their functionality. These systems, such as wireless sensor networks, interact and compose opportunistically, allowing them to create new integrations with potential partner systems. These systems interact through a set of contact-action rules that allow the developer to react to neighboring systems based on their properties or functionality. When a neighbor satisfying these rules is found, the system is notified and can react to the new potential integration. While this approach allows systems to cooperate based on the properties of other systems, it focuses on systems reacting internally, rather than providing behavior to use on another system.

Atzori et al. [7] proposed a paradigm of a social network of smart objects named Social Internet of Things (SIoT) to mimic human behavior. The authors analyzed the types of social relationships between *things* to be: parental (*things* built by the same vendor), co-location and co-work (*things* reside in the same place or cooperate to provide applications), and owner (*things* owned by the same user). The authors of the same project, in [11], also presented an architecture to address network navigability along with service discovery and composition. The architecture is made up of server and objects (the physical devices) as the network elements. The server holds the relationship management module where the selection and setting of the relationships is based on human

control settings along with appropriate interfaces to objects, humans and third-party services. The object side holds an abstraction layer for the device and the social management module for the communication between the device and the server. While our work utilizes inter-*thing* relationships similar to Atzori's work, it goes beyond enabling general communication and links one or more *thing* services to a mobile app, affecting the dynamic behavior of that app based on user interactions.

Lee [21] tackled feature interoperability aspect in the dynamic development of software applications. A feature (a software entity that is visible to users of a system) can collaborate with other features that are not conceived in the application's original design or when it is deployed. The authors proposed a model-driven approach called Dynamic Feature Deployment (DFD) to support the seamless integration of new features and changes to an application's configuration at runtime. DFD is an encoded feature configuration knowledge embedded into the deployed features so that they know their composability. As a new feature model with a changed configuration can be deployed at runtime, features can recognize new features after deployment and can manage configurations depending on a currently bound feature model. The authors also introduced a software model that controls the interactions between the different available features. Our work is different from DFD in that no configuration management is used or needed; rather, it is a dynamic IoT formation involving an app and *things* in a smart space. Additionally, the developer of the mobile app as a *thing*, and the app itself during runtime, does not know specific instances of the *thing* services (that would correspond to DFD features) a priori.

In the apps-as-things demo paper [12], the authors presented a soccer match demo scenario involving media appliances and mobile app *things* (the World Cup Scenario). In this demo, the *things* and app tweet out keywords about their identity, capabilities, and interests. At the same time, they receive these tweets from the other devices, which are parsed to learn about the APIs and services available to the smart space. When these keyword tweets are received, they are compared semantically to the *thing's* own capabilities/interests to identify the potential for new meaningful interactions. In the case of the DVR and mobile app, once a correlation is formed between the "TV" and "recording" capabilities of the two *things*, the DVR passes its recording API to the app. The app then uses this information to adjust its UI layout (adding buttons, notifications, etc.) to allow the user to trigger the new functionality. Such UI presents itself in the form of a "Chromeless" [5] web view, received from the DVR, allowing the user to control it directly from the app. Receiving all capabilities and services provided by *thing* mates provides a *thing* with all the information it needs to form new meaningful interactions. However, in the case of a mobile app *thing*, this also adds much effort for the app developer to manually decide out how to integrate such received capabilities and services into the app logic and interface. In this paper, we present an extension to the Atlas thing architecture [19] targeting mobile app developers, allowing a mobile app to react to opportunities in the smart space without specific intervention or foresight from the app developer.

2.1 Atlas Thing Architecture

As noted earlier, we base our framework on our Atlas thing architecture [19]. The architecture takes advantage of a *thing's* OS services to provide new capabilities a *thing* needs in order to engage in ad hoc interactions and interconnections. The architecture utilizes the specifications of the IoT Device Description Language (IoT-DDL) [9, 16], which is a machine- and human-readable digital description and metadata that is loaded to the *thing* and describes it in terms of its inner components, attachments, resources and the services it offers. The *thing* then discovers its own identity and capabilities, generates services, and formulates APIs to these services to be announced to other space entities. A *thing* in a smart space engages with *thing* mates through a set of information- and action-based interactions. Information-based interactions (referred to as tweets) enable a *thing* to announce its identity, capabilities, and APIs to thing mates. Action-based interactions include management commands, lifetime updates, and configurations from authorized parties as well as applications that target the *thing*.

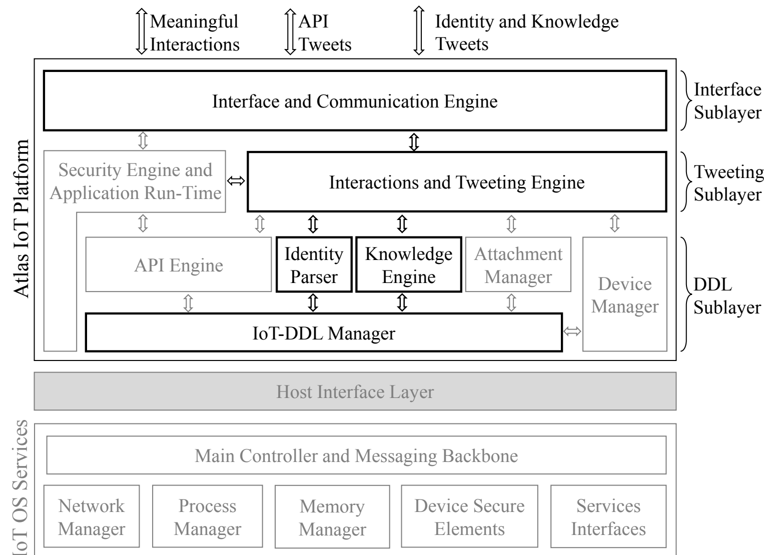

Fig. 1. Atlas Thing Architecture.

The inter-thing relationship programming framework proposed in [17] broadens the social IoT *thing*-level relationships proposed in [7] with service-level relationships that logically and functionally show how the *things'* services may tie to build applications. Such service-level relationships extend this limited and restricted set of relationships with a new set of concrete relationships for a wider class of applications. In this paper, the *things* dynamically detect the opportunistic relationship with their mates' services and utilizes the framework to describe such application in terms of the different primitives and operators. The created application is governed by a set of semantic rules that evaluate correctness and guide execution. Through the mounted architecture and the uploaded IoT-DDL, the *thing* dynamically builds runtime programmable representations for the offered services and relationships and generates services along with the appropriate APIs to them.

Due to space considerations, we do not describe the full scope of the project but focus on the layers that support our framework. The Atlas IoT platform layer of the architecture, illustrated in figure 1, focuses on the descriptive and semantic aspects of *things* to better enable *thing* engagement and programmability. The DDL sublayer configures the architecture according to the IoT-DDL for just-in-time API-ing, identity and device management, own- and learnt-knowledge management. A significant requirement we address is to allow *things* to understand and prepare for new meaningful interactions introduced by a smart space, with minimal intervention required by the user. Such an ability empowers a *thing* to discover new social relationships with the smart space and assist the user in discovering new possible relationships. In the tweeting sublayer, the *thing* builds its own tweets to describe what it is, what it does, and what it knows to the other mates. The *thing* then analyzes the tweets announced by its mates. The discovery of social relationships through ad-hoc social interactions (tweets) which enables the discovery of semantic similarity and affinity between *thing* mates is accomplished in the tweeting sublayer with the help of WordNet [23, 30]. WordNet is a lexical database that groups the English words into sets of synonyms words; measures the semantic similarity between them; and records relations between such sets. Our architecture also utilizes the Inverse Document Frequency measure (IDF) [27], which is a numerical statistic intended to reflect how unique and important a word is in a corpus.

3 REDESIGNING MOBILE APPS TO BE THINGS

Image a user with a mobile app entering a smart space full of smart *things*. For example, a soccer fan with a sports mobile app approaching his media center in his living room (the World Cup Scenario described in section 2), or a travelling businessman with productivity apps arriving at an airport lounge. In this age of IoT, shouldn't he expect new kinds of interactions and suggestions for future or new engagement opportunities? And shouldn't he expect that his mobile app UI update itself to catch and bring awareness to these opportunities? Further, if these scenarios are possible, how long would it take him to notice these new opportunities reflected in his mobile apps?

For the above situation to be realizable, a mobile app as a *thing* must react to the capabilities of a smart space, and change within the context of its own interface and functionality. This calls back to *introspective programs* [8], or programs that "self-reference" their own information. However, instead of performing tasks like copying itself or print out its source code, a mobile app must use this information for far more practical purposes, such as adjusting their services, appearance and functionality on the fly to take advantage of and utilize the services of another smart *thing*. Before such modifications can occur, two pieces of information must be known: the target functionality (from the smart *thing*), and the input data or control to give it (from the mobile app). Once both of these are known by one of the devices, the interaction can occur.

Receiving potential functionalities from the *thing* through the transmission of its APIs, as was done in the apps-as-things demo [12], provides an app with all the information it needs to form new meaningful interactions. However, this method also leaves a lot of work for the app developer, in terms of determining how to integrate this new behavior and display it to the user. If the developer does not know exactly which smart *things* to support, it becomes impossible to determine what context (when and where in the app) an interaction should be available in. In this manner, the app developer may want to leave some "placeholder" space for a new interaction's UI elements; however, anticipating the extent of the requirements (mainly where to place this placeholder and how to link it logically with the created new behavior) would prove difficult. Instead, a UI in this manner would likely find most use in pop-ups (such as a toast notification [4]), or a new interface in its entirety, such as the Chromeless web browser view mentioned above, where the context of the interaction can be entirely contained. This moves the information requirement to the smart *thing*, which can now receive input data through its own interface, using the mobile app only as a display. Unfortunately, in these cases, this context is disjointed from the rest of the interface: the UI exists "on top" of the app, and cannot easily interact with or extend the app's developer-created elements.

True integration of a *thing's* functionality into a mobile app, therefore, becomes difficult. Continuing with the DVR scenario described in section 2, the app knows what program the user wants to record, but it still needs to know where and how to send that contextual data—a difficult situation when the developer lacks prior knowledge of the received DVR API. Many of the interface decisions would have to be made/provided by the *thing* (i.e., the Chromeless web view), although its knowledge about the app is limited in a similar manner (e.g., it must ask again what program to record). If the developer only knows about what information his application can provide to potential *things*, how can the features of these *things* be more closely integrated?

We argue there is more information that can be used by the developer. In a general sense, the developer also knows where the data comes from (such as a specific UI element), and, by extension, how this data could be used in a new relationship with an interested *thing*. In a situation targeting a specific smart *thing*, an app developer would indirectly identify this "interaction-capable" information to pass to the known API. For example, the developer of the TV app knows that each row in the list of TV programs—as shown in figure 1—offers a unique set of values (e.g., the channel number and air time) that is needed to control the DVR's functionality. When the target functionality is unknown, the developer can still identify this information, effectively saying, "the user is interested in a specific TV channel," compared to "the user is interested in TV channels in general," as in the

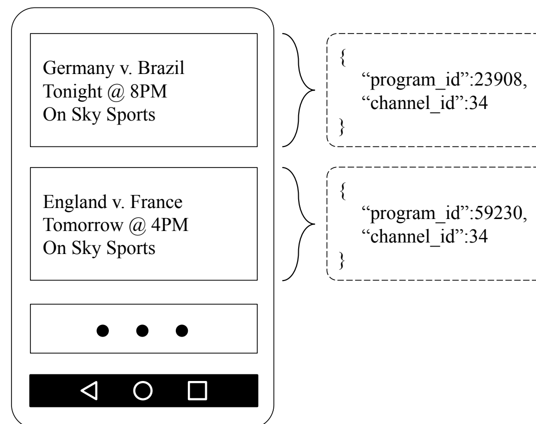

Fig. 2. Independent groups of information available from different elements of a mobile app.

system described in section 2. The mobile app provides the smart *thing* with the input data it needs, without requiring the *thing* to collect the data itself, or know when the data should be used to invoke its service.

As mentioned above, the *thing* does not know when to use this data; it needs the context of the interaction, or where in the app's logical flow this data will enter and be made available. In the case of a mobile app with many views and functionalities, such context is critical in enabling meaningful interactions; at times, a piece of data may not always be relevant or even accessible, depending on the state of the app and the actions of the user. Conveniently, knowing where the data comes from provides the contextual information: the source UI element is tied to a specific point in the app's logical flow. For example, each list row in the TV app provides a complete set of information, and implies that the data is only available when the list view is active (browsed over by the user); the context of an interaction is contained within each specific row. This is illustrated in figure 2: each row is backed by data (a channel and program "ID") needed to trigger an interaction with a smart *thing* (the DVR). More TV programs may exist in the list, but are contextually unavailable; they are not visible to the user and therefore cannot be used to invoke *thing* functionality.

Collecting and presenting this per-component information, however, poses a significant challenge to the developers of both *thing* devices and mobile apps. Making such fine-grained interaction a reality will require mobile app developers to truly consider the role of their app in an IoT system, and *thing* developers to support a wide range of potential interactions. Through a new programming construct introduced in the next section, we aim to limit this apparent complexity by reducing the requirements placed on the mobile app developer, where a mastery of IoT should not be required to create an app with *thing* capabilities. We also remain mindful that a solution should impose only minimal changes to the app development process; even with IoT knowledge, app developers should not need to go out of their way to make their apps *things*. Rather, the capabilities introduced should exist as a first-class citizen within the standard development ecosystem.

4 ACTIONABLE KEYWORDS

For a mobile app to realize its functionality as a *thing*, we utilize the information described above and reverse the roles of a mobile app and normal *thing* devices. Rather than search for potential interactions through service/API information broadcast from *things*, the mobile app advertises its available input data back to the smart space. A *thing* can then match this input against its available services, discovering new potential interactions that can

be shared with the mobile app. In addition to allowing the mobile app developer to create an interface for the broadcast inputs, this also allows *thing* devices to determine the actual matches, which we believe is a more flexible and realistic alternative to requiring the app developer to account for all potential *thing* interactions.

To represent this input information and facilitate the link between the logic of a potential interaction and the mobile user interface, we introduce the *actionable keyword* (AKW) concept and programming construct. An AKW—as shown in figure 3—is a structure present within a mobile app that contains a description of some input data, a set of keywords to represent the purpose/source of that data, and information on the UI elements that data is tied to (the context of the potential interaction). Information on these AKWs is then broadcast to the smart space, where *thing* devices compare keywords and data types against their available services. If a match is found, that service's API is sent back to the app to be invoked by the user. Note that a single AKW may represent multiple instances of such a UI context (like the rows of a list view); the actionable keyword represents the *type* of information available, not a specific piece of data—all instances of that data are valid inputs to a *thing* service that matches with the AKW.

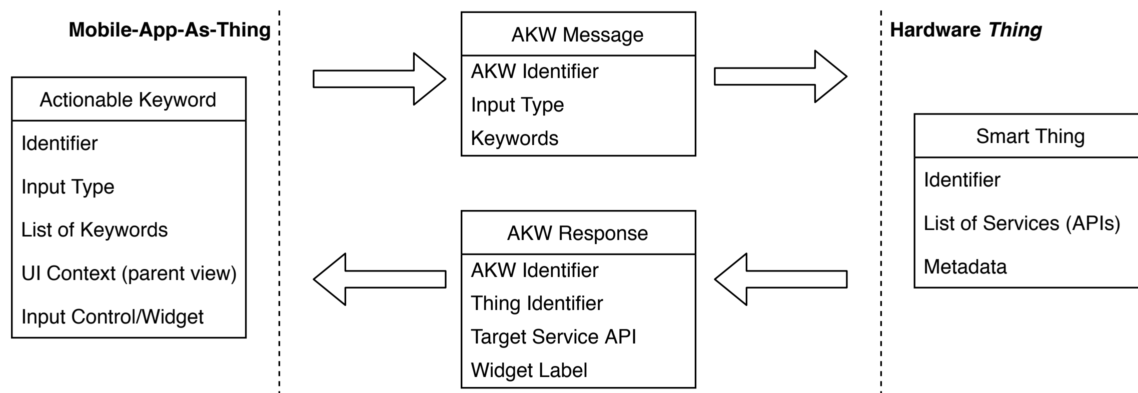

Fig. 3. The *actionable keyword* programming construct, along with the messages passed between app and *thing*.

The above situation highlights a pressing issue: once a relationship is formed through an AKW, *what* data should be used, and *when*, to invoke the *thing* device's functionality? To solve this, an actionable keyword also represents an input control (*widget* [1]) within its context that allows the user to trigger the formed relationship. The developer defines a "placeholder" button that is initially blank and hidden—it is not known what service, if any, will respond to the AKW. Once a relationship is established, the button pops into view with a *thing*-specified label. At this point, tapping the button invokes the received *thing* functionality with the data from the relevant UI context as input.

To illustrate this entire process (as shown in figure 3 and 4), consider the TV guide scenario from section 3. The app developer decides to expose upcoming TV programs to the smart space. The developer decides to transmit the program ID and channel ID of each (see figure 2), describing this data with the keywords "TV", "channel", and "program". The developer then specifies the list's row layout as the UI context, and adds a placeholder button within it. This information completes the definition of the actionable keyword, which can be broadcast to the smart space during runtime. Within the Atlas architecture, this information is sent out as a message called a *tweet*, along with existing tweets describing the app's identity and general capabilities [18]. These AKW tweets can then be evaluated by other *things* in the smart space, searching for potential matches against their available services.

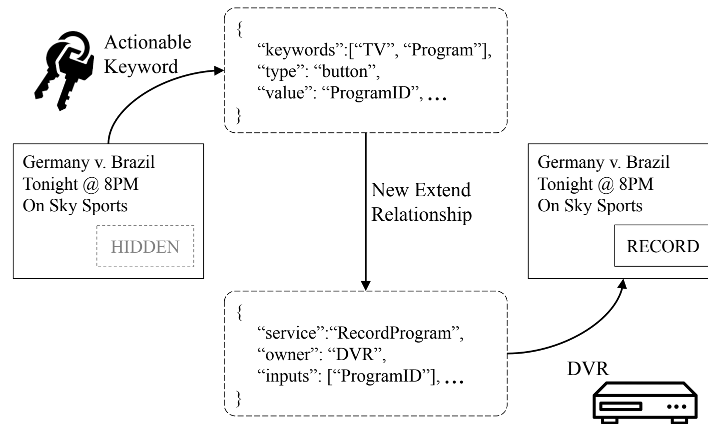

Fig. 4. The interaction between a button's actionable keyword and the DVR service.

The DVR *thing* finds a match and responds to this AKW to create a new relationship, providing information about itself, its recording service, and a new label for the placeholder button within the app. Using the Atlas architecture, the mobile app concatenates its ability to provide information with the functionality offered by the DVR (recording a TV channel) through an *extend* relationship [17]. Once this relationship is established, the app enables the placeholder button in each row of the list view, changing their labels to "RECORD". When a record button is tapped, the service provided by the DVR is invoked with the channel and program info from the button's context. Note that multiple services may vie for an AKW, but only one may attach and create a relationship—this, however, is not permanent; the bound service may be removed or disabled, allowing another to match.

In the given scenario, all information needed by the DVR service is available from the app immediately, and the user can complete the interaction with the single tap of a button. The interaction involves a *thing* requiring a specific input type, receiving the relevant AKW, and instructing the mobile app to integrate its service. However, this kind of back-and-forth may not always be able to satisfy a *thing* service's requirements, especially with more complicated services, such as those requiring multi-stage interactions or user confirmation. Handling such interactions is likely to drastically increase the complexity of the solution, therefore, the proposed systems in this paper will focus only on "simple" interactions utilizing a single button, to lay groundwork for future improvements supporting these complexities, as described in section 8.

5 IMPLEMENTATION

The initial design and implementation of our Atlas architecture extension and actionable keyword programming construct targets Android mobile applications. This includes a custom UI component to represent an AKW, an Android service process to manage active keywords and communicate with the smart space, and a superclass of the Android *Application* class to handle data exchange between these parts. These components allow the actionable keywords to perform their functionality (broadcasting, listening, and updating the UI) independently; the runtime behavior of individual layouts or keywords within an *Activity* does not need to be managed.

The custom UI component, called an *ActionableLayout*, is a subclass of the Android *ViewGroup* class (similar to the standard layout types like *LinearLayout* [2]) intended to wrap around an existing layout or view component. This component represents the context of an actionable keyword—the elements within the *ActionableLayout* represent the data that can be offered to the smart space. Each *ActionableLayout* requires the following elements:

1) a list of keywords, 2) a representation of the data it provides, 3) an associated button component, and 4) a data formatting listener function. An example Android layout XML tag for an *ActionableLayout* is shown in figure 5.

The given keyword list and data representation are used as provided to broadcast the actionable keyword to other things, while the button and data formatter are used internally to facilitate the app's dynamic behavior in reaction to a *thing* responding to the AKW. The button component, as described in section 4, is a child component of the *ActionableLayout* that is initially hidden, but becomes visible and enabled once a *thing* responds to the actionable keyword, as shown in figure 6. The responding *thing* is given some control over this component through its label (the *thing* service may specify the text that should appear within the button). This allows the new functionality to be presented to the user without requiring specific knowledge from the original app developer. The data formatting function, the final requisite part of an AKW definition, replaces the concept of the *onClick* listener for the *ActionableLayout's* button; while button clicks are handled internally, they collect the data to be sent to the appropriate *thing* service by referencing this developer-defined data formatting function (see figure 5). This allows the developer to pull the data from their application logic and format it as they specified in the *ActionableLayout* parameters, without needing to manage the AKW display logic.

```
<ActionableLayout
    android:id="@+id/myAKW"
    app:button="@+id/abutton"
    app:data_format="[ProgramID, ChannelID]"
    app:keywords="TV, Show, Program"
    app:onFormat="onFormatProgram">
    ...
    <Button android:id="@+id/abutton" />
    ...
</ActionableLayout>
```

```
public JSONObject onFormatProgram(View v) {
    Program p = (Program)v.getTag();

    JSONObject o = new JSONObject();
    o.put("ProgramID", p.id);
    o.put("ChannelID", p.channel);

    return o;
}
```

Fig. 5. An *ActionableLayout* XML definition, and its companion formatter function.

When an application utilizing actionable keywords is running, a supporting Android service becomes active in the background. This service performs most of the traditional Atlas architecture functionality, such as listening for and responding to tweets. In the context of actionable keywords, this includes sending keyword info, handling interested *things*, and invoking *thing* services. *ActionableLayouts* register their AKW with the service if needed (multiple instances of an individual layout refer to the same AKW and only register once), and are notified by the service to enable their button upon receiving a response.

5.1 Responsiveness

Even after an actionable keyword is matched and the mobile app's interface updates, nothing can occur until the user sees the change and interacts with the new UI elements. User understanding is an important part of the actionable keyword concept, especially when the signs of an AKW are mostly "invisible" before it is matched; a user will likely only notice a new interaction after it has formed and the UI elements appear. Between the required communication and keyword matching time, any modifications to the app's UI will occur with some level of delay from the time the AKW is broadcast. Depending on the amount of delay, the user may miss interaction opportunities by navigating throughout the app to quickly, or become confused when a UI element has changed since the last time it was viewed.

App responsiveness and user awareness, therefore, are critical to the functioning of actionable keyword interactions. The presentation of an AKW before and after it is matched must be carefully considered to ensure

using such an app does not confuse the user or feel clunky or sluggish. Mainly, this involves handling what happens when an AKW's button becomes visible during a match, in terms of latency and user perception. For example, the sudden appearance of a UI element might confuse the user: where did such a button come from and why? One possible solution would be to use a progress bar or other element to gradually replace the hidden button (see figure 6), raising awareness to the use of an ongoing search for AKWs. This would hint to the user that something may occur in that area of the UI. However, this may also cause confusion if the delay before a match is too short and the progress bar appears only briefly, unless a limit is set on how fast the bar may progress. Optimizing the behavior of a progress bar in the MAAT UI design is important; however, in the current implementation iteration, we have adopted a simple "pop-in" design.

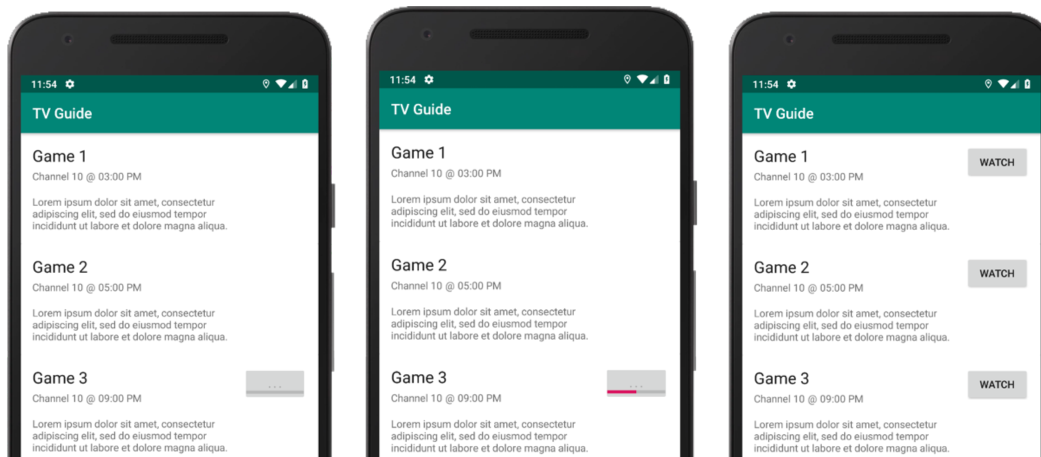

Fig. 6. The different states (before: left, during: middle, after: right) of the app UI as an actionable keyword is broadcast and tied to a service. "Game 1" and "Game 2" show a simple "pop-in" button, while "Game 3" shows a "search" progress bar.

For the initial implementation of the actionable keywords programming concept, we chose to focus on the latency between an AKW broadcast and match, to minimize the user's perceived delay. With a small enough latency, the button will appear during navigation transitions or within reasonable human reaction time, giving the illusion that the button is seamlessly integrated with the original app. We explore this responsiveness and latency between apps and things further in section 6, with a set of experiments that record these metrics under varying conditions.

5.2 IDE Plugin

Using actionable keywords does require developers to somewhat modify the way they advertise and search for services. The *thing* developer (the DVR manufacturer, continuing on our example), must look not only for the type of app/*thing* ("TV-related") their service can utilize, but also for the specific data their service requires ("TV Program ID" or similar). On the app side, the developer (or their IDE) must consider which elements of the app they desire to make available to a smart space, and then adequately describe what the element is offering such that a *thing* can recognize a potential for interaction when it is there.

Both requirements may be overlooked by a developer who is not engrossed in IoT. In the original case (see section 2), the developer provides keywords and capabilities for the app as a whole, probably with little need to update. Now, however, AKWs can be specified in many components across the app and may be added or removed

as features change or the UI is altered. This increases the burden on the developer, who must add these actionable keywords (both descriptively and as part of the UI), based partly on their imagination of what information might be useful to a smart space.

To facilitate the creation of actionable keywords, and reduce this burden, we implemented an AKW plugin targeting the Android Studio IDE [15]. The plugin consists of a series of dialog windows guiding the developer through the creation of AKWs, as illustrated in figure 7. The plugin provides support in: 1) creating the *ActionableLayout* with the appropriate fields, 2) choosing the keywords and data format, and 3) implementing the data formatter. The developer starts by selecting a layout component that should be wrapped by an *ActionableLayout*, and initiating an Android Studio intention action [14] provided by the plugin. An *intention action* (or "lightbulb") is an IDE feature that provides localized warnings or suggestions within an active source file. This is commonly used by developers to fix errors or import dependencies.

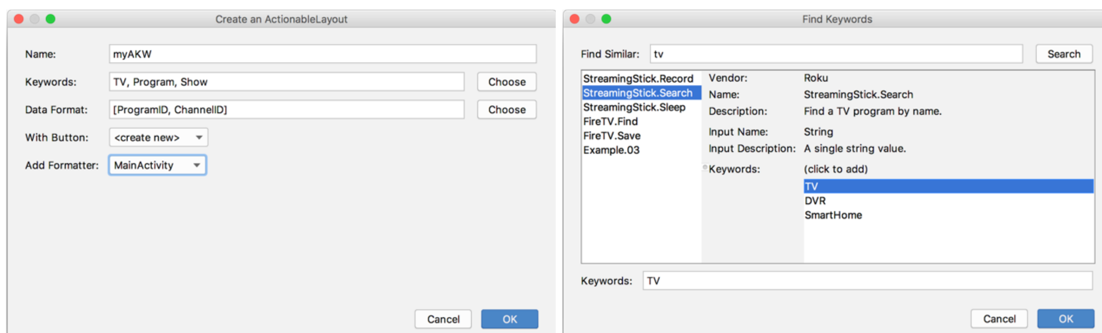

Fig. 7. The AKW IDE plugin interface.

Invoking the AKW intention creates a dialog with fields for the required data. The fields for keywords and data format may create an additional dialog, allowing the developer to search a repository of existing keywords (described in the next section) within the plugin and directly select from this information. Accepting the dialog wraps the selected layout element in an *ActionableLayout*, sets up any needed references, and creates a skeleton function for the data formatter within the relevant activity class. This skeleton function creates a JSON object with the appropriate key names for the chosen data format, only requiring the developer to fill in the values (see figure 5).

5.3 Sourcing Actionable Keywords

As mentioned above, the developer not only has to describe the information being offered, but also must provide the information in a form usable by the *thing* service. For example, the app developer may provide program information in a tuple of channel number and air time, while the DVR developer expects this as a vendor-specific "program ID" number. Due to this mismatch, the app and *thing* would not be able to directly form a new relationship—the *thing* service does not expect the data in the format given by the app. In addition to providing accurate keywords, a mobile app acting as a *thing* must also cooperate with the behavior of potential *thing* interactions. To increase the likelihood of finding a compatible interaction, presentation of data from the app can be facilitated with a mixture of IDE intervention and a repository of existing standards and *thing* descriptions.

One way to reduce potential compatibility issues could be the suggestion and use of standardized object representations: within the actionable keyword, the developer may specify the schema of their data, allowing *things* to better identify the format of the data they are interested in. For example, the app developer may want

to specify the encoding of their TV channel listings as XMLTV [31]—a standard object format for describing TV programs. The DVR developer can then use this information (available from the AKW) to know the app is providing TV information, rather than just seeing a tuple of plain integers. Specification of an existing object representation also reduces the effort needed by the app developer to determine what data their app may broadcast and encourages *thing* developers to support these common formats—increasing the chances of new interactions being formed.

However, an object format may not always be sufficient for all types of *thing* services; an interaction's data may have many standardized representations, or not fit well into what is available. To cover these situations, keywords are instead supported by a central repository of information obtained from existing IoT-DDLs: the human-readable manufacturer descriptions that exist on Atlas thing devices. Because these descriptions already contain the keywords and inputs of existing *thing* services, they can easily be connected and indexed as a database of potential *thing* interactions that will work directly with a MAAT application. This repository (a web-accessible database that indexes vendor, service, and input descriptions) then allows for developers to search (directly in the IDE plugin) through existing services with their own terms, but choose keywords and data formats that are guaranteed to work with their app. This interface can be seen in the second image of figure 7. The availability of this information can help influence the developer in their choice of keywords, their structuring of the app for unknown interactions, and their providing of data to best support a wide range of IoT devices. Note that the repository does not need to inform the developer exactly which *things* should be supported in their app, but instead offers insight on the potential for *thing* interactions.

Such a repository may also benefit *thing* vendors: the capabilities of their devices become easily known, and the information available may influence their service definitions, helping to achieve greater potential for interaction with mobile apps. Continuing the above example, an app developer would likely choose a data format that is supported by multiple TV-related devices in the repository for maximum compatibility. The common usage of a single format may influence a vendor when creating a similar device; by utilizing this interface, the *thing* vendor increases their capability for interaction with various TV-related mobile applications.

6 PERFORMANCE EVALUATION

In this section, we adopt two main metrics for evaluating our implementation of the MAAT idea. The first is the latency of discovering new engagement opportunities between a mobile app and other IoT *things* in a smart space. The second is the responsiveness of the behaviorally-altered mobile app's UI in reaction to these changes.

While metrics such as battery usage and processing power are important in all mobile applications, the main factors for a true evaluation of feasibility and usability in an app-as-a-thing scenario are different. We describe a set of experiments to benchmark the time elapsed in the different phases of an actionable keyword's lifecycle. We believe minimizing this elapsed time is critical to user experience: because AKWs are tied to specific interface elements (and therefore an *Activity*), relationships may be formed on the fly as the user navigates through the app. A delayed change happening after the user has been looking at the UI could be confusing or annoying; therefore, the time required to process a keyword and form the relationship/changes should be as small as possible. There is no specific duration that constitutes "too long" in waiting for a response (the appearance of an AKW), especially when the user's attention is likely on the core features of the app. In these experiments, we consider durations up to one second to be an acceptable response time. Within this range, responses occurring within 100 milliseconds are perceived to be instantaneous, while 1 second is a noticeable but well within tolerable times [20, 24]. It is also important to note that the default activity transition time in Android is defined as 220 milliseconds [3]. Therefore, any interaction occurring within this time (before the transition completes) will not require any additional delay to display a new capability to the user.

The experiments detailed below break the total response time into three separate measurements: 1) the opportunity discovery latency, or the transmission time for sending a tweet to a *thing* device plus that for sending the response to the app, 2) the keyword match time, or the time required by a *thing* to compare an AKW's keywords with its own, and 3) the UI update time, or the time required by the Android app to redraw the AKW interface elements. To measure these durations, benchmark programs are deployed on two representative *thing* devices. The first is the app-as-a-thing running on Nexus 9 with Android version 6.0.1 and 2 GB RAM, and the second is an Intel Edison development board with 500 MHz CPU and 1GB RAM. These *things* are connected to the same private wireless network. The first *thing* used in the experiment is obviously the real target device which is the platform hosting the mobile apps, whereas the second *thing* used is a real hardware platform that adequately resembles and represents "real-life" target *things* such as consumer electronics (e.g., the DVR used in our running example). Therefore, it should be feasible to transfer our approach to these real-life target *things* in the future.

6.1 Experiment I – AKW Activation Time

This experiment benchmarks the *activation time* of an actionable keyword; that is, the time between the broadcast of the AKW tweet and the appearance of the interface element after an appropriate match is received. This time is equivalent to the perceived delay experienced by the user as new relationships are discovered between the app and the smart space, as well as the sum of the three measurements defined above. Each of the three components of activation time (see above) were measured using the Intel Edison *thing* device with a single advertised actionable keyword.

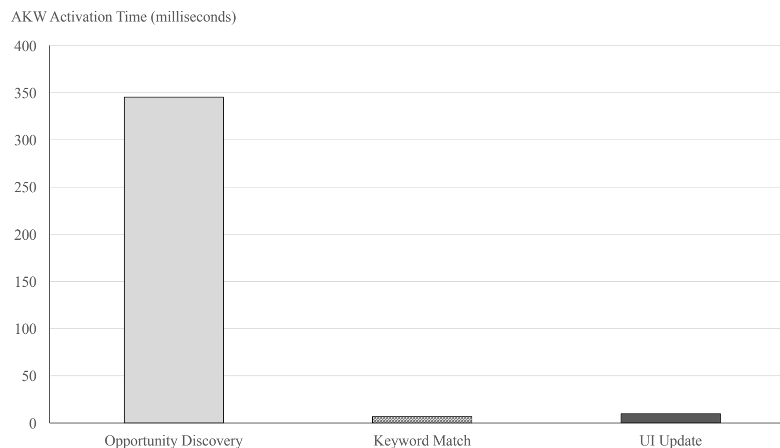

Fig. 8. Segmented activation time for an Intel Edison *thing* interacting with an actionable keyword.

As illustrated in figure 8, the network activity of the opportunity discovery consumes the majority of the time as it depends on the current traffic on the network as well as the network module in use (e.g., WiFi or Ethernet). Both the keyword match and UI update times are minimal in comparison; together, the activation time on an Intel Edison devices would result in an in-app delay of about 150 milliseconds after the activity transitions into view.

6.2 Experiment II – UI Update Time for Multiple AKW Instances

This experiment benchmarks the time needed to redraw the Android user interface as the number of active AKW elements increases. Multiple instances of the same *ActionableLayout* are used, such that they are all activated at

the same time when the appropriate AKW response is received. This case reflects a situation like the TV guide app from section 1, where an AKW is embedded into a list view, and each row represents an instance of the same AKW information. This UI update time is critical to a user's experience, as it directly affects the performance of the app. Too long a delay could cause the app to appear unresponsive.

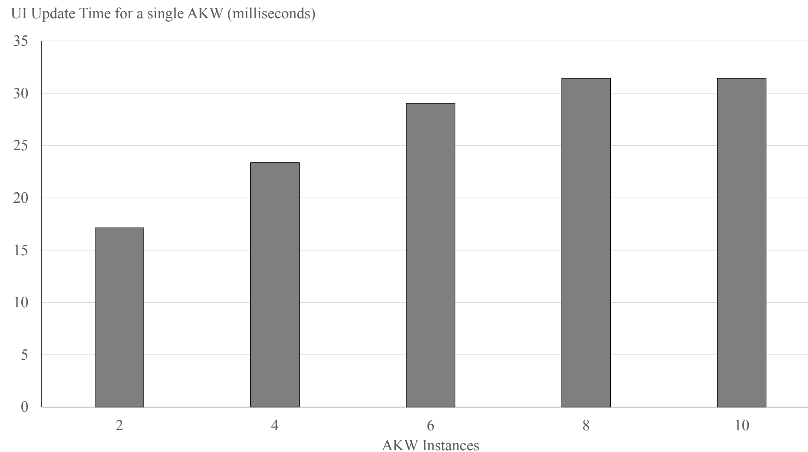

Fig. 9. The UI update time for multiple instances of the same AKW.

As illustrated in figure 9, the update time increases as the number of keywords increases, but levels off at around 8 keywords. This is because rows 8-10 reside off-screen, hidden by the list view layout; they are only updated once they are visible to the user. If an element is active but not visible, the Android OS will update it at a slower interval. Overall, the total update duration is low, taking approximately 2 frames (about 33 milliseconds at 60 frames per second) to display the new capabilities. Note that this is not active rendering time; it is simply the time from when a redraw is requested to when it actually occurs.

6.3 Experiment III – Activation Time for Multiple AKWs

This experiment benchmarks the effect on total activation time when multiple AKWs are being activated across multiple *things*. First, we consider the effect the number of "background" tweets has on activation time; that is, actionable keyword messages (from this app or others) that will not be matched with a *thing* in the current smart space. To do this, an Intel Edison was configured to match a single AKW with the app. At the same time, as illustrated in figure 10, up to 19 other AKWs are broadcast—these simulate "unrelated" keywords that are not compatible with the *thing*. The *thing* device must receive and compare these keywords before rejecting them: they do not affect the app, as it will not configure a new relationship with them.

The additional network load has a reasonable effect on the *thing* device, increasing networking time as well as the time needed to compare keywords for a match. The *thing* is able to multithread the receiving and comparing of AKWs, to help limit the effect a large number of active AKWs has on the device.

Next, as illustrated in figure 11, we consider a situation to test the limits of our actionable keyword concept and programming construct. An app with up to 20 active AKWs, all on-screen and visible in a single *Activity*, was configured to match these keywords evenly across four Edison *thing* devices. Because these keywords are received and processed independently, we consider only the total activation time of the entire set; that is, the time between sending the first AKW tweet and updating the final UI element.

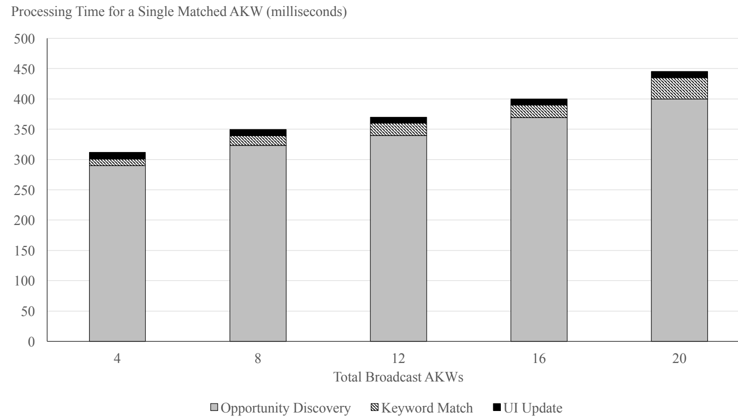

Fig. 10. Processing multiple AKWs when only one will match, on an Intel Edison device.

The total activation time increases steadily as the number of active AKWs increase, but even at 20 elements (completely filling the screen of our Android tablet device), the time remains manageable at about half of a second. We believe that this represents a high number of AKWs for a single interface; although more may exist across an entire application, off-screen instances may not be actively broadcast and update at a slower rate as mentioned above. The overall activation time stays reasonable across a wide range of AKW loads.

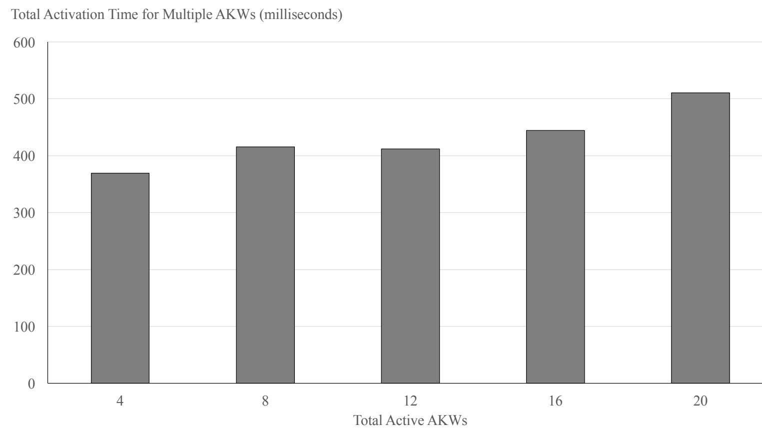

Fig. 11. Enabling a set of AKWs across multiple thing devices.

7 USABILITY EVALUATION

In addition to the performance of the MAAT framework and its interactions with *things*, we also placed emphasis on the functionality and ease of use of the IDE support plugin. In this section, we describe a short study conducted to gauge the usability of the MAAT plugin from an Android developer viewpoint. The study evaluated three usability metrics: effectiveness, efficiency, and satisfaction. These metrics are derived from the ISO/IEC 9126-4 Usability Standard and are defined as follows [28].

Effectiveness represents the accuracy and completeness with which the developers were able to achieve the specified goals. This calculation is reflected in equation (1).

$$Effectiveness = \frac{Number\ of\ tasks\ completed\ successfully}{Total\ number\ of\ tasks\ undertaken}\ x\ 100 \tag{1}$$

Efficiency represents the resources expended in relation to the accuracy and completeness with which the developers were able to achieve the specified goals. Efficiency is measured in terms of *task time* (i.e., the time in minutes a participant takes to complete a task successfully). This can be computed as *time-based efficiency* or Overall Relative Efficiency (ORE), as shown in equations (2). ORE is the ratio of time taken by users who successfully complete a task to total time taken by all users.

Satisfaction represents the comfort and acceptability of the plugin. This is measured through a standardized satisfaction questionnaire that is administered at the end of the tasks.

$$Time\ Based\ Efficiency = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N}\frac{n_{ij}}{t_{ij}}}{NR} \quad \text{and} \quad ORE = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N} n_{ij}t_{ij}}{\sum_{j=1}^{R}\sum_{i=1}^{N} t_{ij}}\ x\ 100 \tag{2}$$

Where:

N	= the total number of tasks (goals)
R	= the total number of users
n_{ij}	= the result of task i by the user j; if the user completes the task successfully, then $n_{ij} = 1$, else 0.
t_{ij}	= the time spent by user j to complete task i successfully, or until quitting.

7.1 Methodology

A laboratory approach was adopted for the usability study, which used eight participants with varying levels of experience in Android development and Android Studio. At the start of the experiment the participant was given a short explanation of the MAAT framework and some time to look over our plugin documentation. The study was divided into two parts; in the first, to measure effectiveness and efficiency, the participants were asked to complete three timed development tasks with increasing complexity (see table 1). The time taken to complete each task, along with any questions asked or errors encountered, was recorded for each. The second part of the study asked participants to complete a survey at the end, measuring satisfaction.

Table 1. Usability study tasks.

Task	Description	Expected Duration (minutes)
1	Create a simple actionable keyword	5
2	Create two independent AKWs in a single *Activity*	10
3	Use list data to make a multi-instance AKW	15

7.2 Effectiveness and Efficiency Results

The results of the study's first part, measuring effectiveness and efficiency, are shown in table 2. Apart from Task 1, each task was completed either within the allocated time or much sooner. The discrepancy with Task 1 can be attributed to the initial unfamiliarity with the MAAT framework in general. While participants understood the initial documentation, some hands-on guidance on the concepts was often needed initially. However, once the developers gained this intuition, few issues were encountered in future use. This is especially prevalent in the times of Task 2, in which the concepts of Task 1 could be directly applied. Task 3, the most complex, required some traditional Android development in addition to direct use of the plugin, but was still usually achieved within the expected time. The participants encountered a total of eight errors, for an average of one error per three tasks. Most errors in Part 1 resulted from the unfamiliarity described above, which were recorded based on the participant's need for direct assistance. The errors from Part 3 usually occurred while "connecting" the Android interface data to the AKW backend.

Table 2. Usability study task results.

Participant	Task 1 Time (# Errors)	Task 2 Time (# Errors)	Task 3 Time (# Errors)	Completed Tasks
1	8 minutes (1 error)	4 minutes (0 errors)	10 minutes (0 errors)	3
2	12 minutes (1 error)	7 minutes (0 errors)	14 minutes (1 error)	3
3	10 minutes (0 errors)	5 minutes (0 errors)	10 minutes (0 errors)	3
4	10 minutes (0 errors)	6 minutes (0 errors)	14 minutes (1 error)	3
5	5 minutes (0 errors)	4 minutes (0 errors)	12 minutes (1 error)	3
6	10 minutes (1 error)	6 minutes (0 errors)	16 minutes (0 errors)	3
7	6 minutes (0 errors)	5 minutes (0 errors)	15 minutes (1 error)	3
8	10 minutes (1 error)	4 minutes (0 errors)	11 minutes (0 errors)	3

All of the participants were able to complete their three tasks successfully; therefore, to calculate effectiveness of the MAAT plugin we consider only the tasks that were completed without error as successful. The resultant effectiveness across the tasks can be seen in table 3. The efficiency of the plugin was computed using the Overall Relative Efficiency. Using the same success metric as effectiveness, the ORE for the MAAT plugin equals (119/214) = 55.6%. This figure is relatively low due to treating tasks that generated errors as unsuccessful, especially considering the need for familiarity as described above. If we discount the errors from Task 1, the ORE jumps to 74.2%. These results highlight the fact that even with the plugin, developers still need some level of knowledge in IoT and the MAAT framework to fully utilize it.

Table 3. Effectiveness of the plugin across tasks.

Task	Computation	Effectiveness (%)	Average Effectiveness (%)
1	4/8	50	
2	8/8	100	70.8
3	5/8	62.5	

7.3 Satisfaction Survey Results

The second part of the survey, assessing developer satisfaction, asked each participant 14 questions ranging from how simple it was to use the plugin, to how useful was the information provided in completing the tasks, to whether workflow would have to be changed to accommodate the plugin (see figure 12). 87.5% of the participants felt the plugin was easy to use, and a similar number felt they could complete their work effectively using the plugin. Importantly, most participants noted that they did not have to alter the way they worked to use the plugin. Only two aspects of the plugin received a weighted score of less than 4.75 out of 6; most participants felt that the plugin did not provide clear error messages or facilitate quick recovery from mistakes. The initial version of our plugin does provide detailed help for many of the situations encountered in the study; this information clearly shows areas of improvement for the next version of the plugin. Overall, however, participants indicated they were satisfied with their use of the plugin.

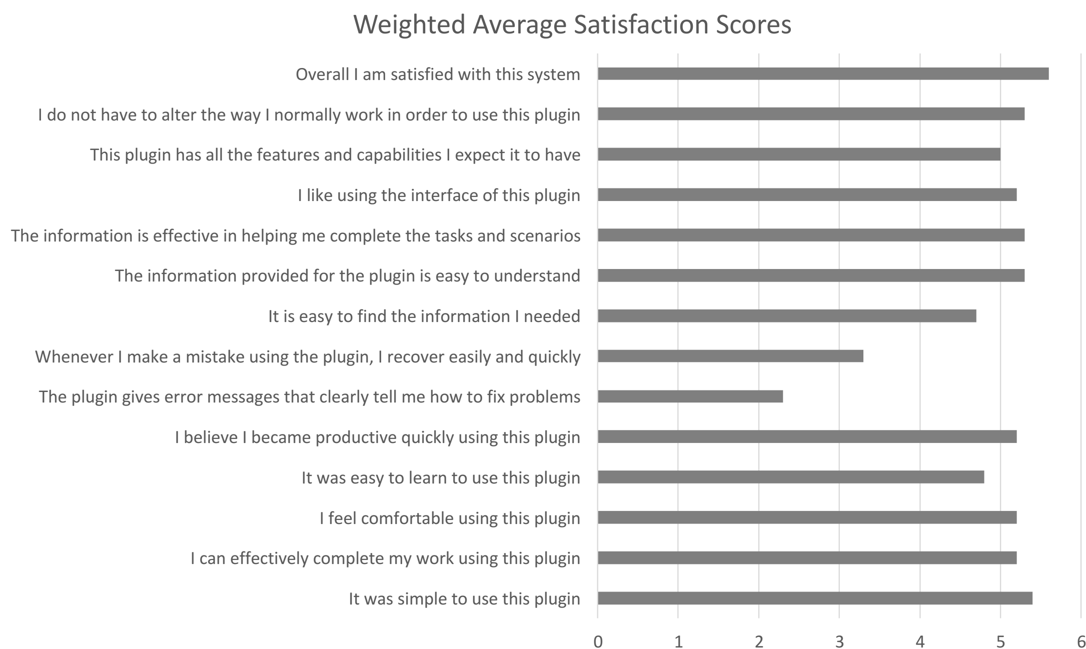

Fig. 12. Satisfaction survey.

8 DISCUSSION AND FUTURE WORK

As mentioned in section 3, we consider only a simple case where an actionable keyword interacts directly with a *thing* service through a single button widget. However, this considers only straightforward situations where the AKW's action maps directly to the invocation of a *thing* API taking an input and giving an output. Introducing a user interface and direct interaction into a *thing* service loop creates potential for more complex functionality to be expressed, such as multiple inputs or "chained" service calls (such as asking for further information, providing branching options, etc.) whose state can be reflected within the app—state which would normally be difficult to convey through interface-less devices in a smart space. Additionally, the capability for user interaction could be

expanded to include other input widgets [1]—such as a dropdown to enable multiple AKW bindings—expanding on the current "invoke a service now" capability brought by a single button.

The evaluation of our work can also expand beyond opportunity discovery latency and app responsiveness to include other performance metrics that may be investigated to further improve and evaluate our work. This may include: 1) accuracy, in terms of how relevant the added widgets are to the interface, and 2) awareness, in terms of how the user understands an impending change that will alter the app and its UI, along with the balance between too much awareness (notifications for every change) and none at all (where the user is surprised by the changes). In this section, we consider potential direct improvements to the current framework and IDE plugin based on the results of our evaluations.

8.1 Minimizing App Developer Overhead

The IDE plugin assists the developer in choosing AKW attributes and in visualizing the smart space landscape. Such functionality, however, moves away slightly from the goal of minimizing the app developer's requisite knowledge and eye for IoT. Choosing common standards and searching through an IoT-DDL repository still places some burden on the app developer in supporting their application as a *thing*. Here, we consider the possibility of deducing and presenting some of this information through the IDE plugin. The plugin, in such a case, could consider the developer's current code and suggest if it is an element that has IoT potential. For example, the plugin may see an Android developer creating a *ListView*, and ask if the data in the *ArrayAdapter* (the data structure providing display information to the *ListView*) could be broadcast to a potential smart space. The plugin may provide a template for this new actionable keyword description (similar to the current functionality, but with some parameters pre-filled) and make suggestions on how to send out the data or modify the user interface. The plugin could, in some cases, deduce the type of interaction from the existing interface elements and the data provider, only needing the desired keywords from the app developer to create this skeleton of an actionable keyword. The chosen keywords may then be used to further refine the internals of the AKW.

At such a point, the app developer only needs to provide two decisions to create a *thing*-ready mobile application: the format of the broadcast data, and what keywords should describe it. This might allow the plugin to create AKW skeletons for many interface elements; a portion would likely remain unused, but only one needs to be recognized and used to create the potential for a meaningful reaction. This, however, creates some issues with the keywords themselves; they must somehow be deduced from surrounding information (such as other interface elements or the repository), or input with some form of developer intervention. Additionally, there are issues with potential overhead depending on the size and complexity of the app interface and chosen AKWs; the number of auto-generated actionable keywords could be large, saturating the smart space with unwanted tweets. The plugin therefore might need some way to prune elements which are unlikely to form relationships before recommending them to the developer.

8.2 Sourcing and Identifying Actionable Keywords

In this paper, we have opted for a design that does not require the use of pre-existing ontologies for actionable keywords, while putting in place elements—using our solution—that promote the gradual development and eventual convergence of a more grass-roots, community-driven ontology. The central repository utilized by the Android Studio IDE plugin plays this role, where initial and future AKWs can be deposited, browsed and used by the developers. As such "organic" development progresses, we envision that broadly available standards would naturally be used in defining AKWs by the developer community (as is demonstrated through TV Program-related AKWs in our running example). We believe our approach of not "putting the carriage before the horse" as we first attempt to realize the mobile app-as-a-thing concept is important at this start phase of the research project.

Although this keyword information will be readily available to developers through the IDE plugin and its central repository (as explained above), actually identifying an AKW with the right level of abstraction can be a challenging task. If a keyword is too abstract (e.g., "TV" or "Time"), it may include many services and this makes it difficult to indicate which service to interact with. If an AKW relates to a low-level service (e.g., "Track Search" or "Retrieve View History"), this might allow the user to control unexpected operations with possible security/integrity issues. As a guideline to address this challenge, a solution such as feature modeling [22] could be adapted for AKW identification.

Feature modeling is the activity of identifying externally visible characteristics of a family of similar products (i.e., a product line or a product family) and organizing them into a model called a feature model. Figure 13 shows part of the feature model for a DVR and a TV guide app. The primary goal of feature modeling is to identify the core features of products and represent them in an exploitable form. We envision such a model could benefit our framework by introducing a separation of two service characteristics: the *AKW* and *Internal Operation* layers.

Features in the *AKW* layer (see the right side of figure 13) are the candidate AKWs, where each of them is composed of a set of operational features. This means that they hide the operational details of the services but capture the visible services provided by a *thing*. A feature in the *Internal Operation* layer are for the fine-grained operations required to provide the AKW services. As shown in figure 13, the visualization of the hierarchical structure among features is one of the key benefits for adapting the feature modeling. The *thing* and app developers may use this visualization to decide which features to target and expose with AKWs and which features to hide.

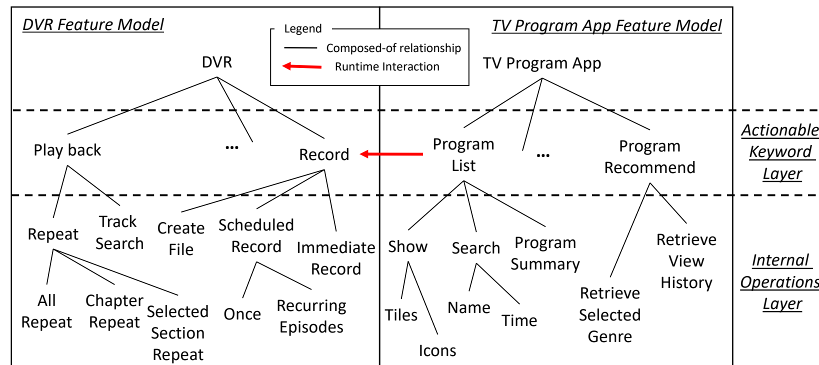

Fig. 13. AKW identification with feature models.

9 CONCLUSION

There is a great potential for mutual opportunistic interactions between *things* of an IoT in a smart space and traditional mobile apps. We provided examples to support this argument. To enable such an important possibility, we propose a redesign of the structure of mobile apps along within our Atlas thing architecture to realize the concept of Mobile Apps As Things (MAAT). We also propose *actionable keywords*—a dynamically programmable description—to enable the dynamic and opportunistic *thing* to *thing* interactions. We presented an Android based implementation of MAAT which included programming support for Android developers in the form of Android Studio plugin. We conducted a performance evaluation study to test the responsiveness of our ideas and implementation and have clearly demonstrated this feasibility. We also conducted a usability study of the plugin with end users who are reasonably experienced Android developers and learned what seems to be working well and what needs to be improved.

REFERENCES

[1] Android Input Controls [Online] 2019. google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/.
[2] Android Layouts [Online] 2019. developer.android.com/guide/topics/ui/declaring-layout.
[3] Android Source Tree - config.xml [Online] 2019. android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/config.xml.
[4] Android Toast Overview [Online] 2019. developer.android.com/guide/topics/ui/notifiers/toasts.
[5] Android WebView [Online] 2019. developer.android.com/reference/android/webkit/WebView.
[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2011. SIoT: Giving a Social Structure to the Internet of Things. *IEEE Communications Letters* (11 2011). https://doi.org/10.1109/LCOMM.2011.090911.111340
[7] Luigi Atzori, Antonio Iera, Giacomo Morabito, and Michele Nitti. 2012. The Social Internet of Things (SIoT) – When social networks meet the Internet of Things: Concept, architecture and network characterization. *Computer Networks* 56 (11 2012).
[8] Paul Bratley and Jean Millo. 1972. Computer recreations. *Software: Practice and Experience* 2, 4 (1972), 397–400. https://doi.org/10.1002/spe.4380020411 arXiv:onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380020411
[9] Chao Chen and Abdelsalam Helal. 2009. Device Integration in SODA Using the Device Description Language. *2009 Ninth Annual International Symposium on Applications and the Internet* (2009), 100–106.
[10] Geoff Coulson, Gordon Blair, Yehia Elkhatib, and Andreas Mauthe. 2015. The design of a generalised approach to the programming of systems of systems. https://doi.org/10.1109/WoWMoM.2015.7158188
[11] Roberto Girau, Michele Nitti, and Luigi Atzori. 2013. Implementation of an Experimental Platform for the Social Internet of Things. *Proceedings - 7th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2013*, 500–505.
[12] Sumi Helal, Ahmed Khaled, and Venkata Gutta. 2017. Demo: Atlas Thing Architecture: Enabling Mobile Apps as Things in the IoT. 480–482. https://doi.org/10.1145/3117811.3119856
[13] IFTTT [Online] 2019. ifttt.com.
[14] IntelliJ IDEA Intention Actions [Online] 2019. www.jetbrains.com/help/idea/intention-actions.html.
[15] IntelliJ Platform SDK DevGuide Part I - Plugins [Online] 2019. www.jetbrains.org/intellij/sdk/docs/basics.html.
[16] Ahmed Khaled, Abdelsalam Helal, Wyatt Lindquist, and Choonhwa Lee. 2018. IoT-DDL—Device Description Language for the "T" in IoT. *IEEE Access* PP (04 2018), 1–1. https://doi.org/10.1109/ACCESS.2018.2825295
[17] Ahmed Khaled and Sumi Helal. 2018. A framework for inter-thing relationships for programming the social IoT. In *IEEE 4th World Forum on Internet of Things (WF-IoT 2018)*. 670–675. https://doi.org/10.1109/WF-IoT.2018.8355215
[18] Ahmed Khaled, Wyatt Lindquist, and Sumi Helal. 2018. Service-Relationship Programming Framework for the Social IoT. *Open Journal of Internet of Things (OJIOT)* (2018), 35–53.
[19] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. 2006. Atlas: A Service-Oriented Sensor Platform: Hardware and Middleware to Enable Programmable Pervasive Spaces. *2006 31st IEEE Conf. on Local Computer Networks* (2006), 630–638.
[20] Carine Lallemand and Guillaume Gronier. 2012. Enhancing User eXperience during waiting time in HCI: Contributions of cognitive psychology. *Proceedings of the Designing Interactive Systems Conference, DIS '12*. https://doi.org/10.1145/2317956.2318069
[21] Jaejoon Lee. 2013. Dynamic feature deployment and composition for dynamic software product lines. *ACM International Conference Proceeding Series*, 114–116. https://doi.org/10.1145/2499777.2500717
[22] Kwanwoo Lee, Kyo Kang, and Jaejoon Lee. 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. *7th International Conference on Software Reuse: Methods, Techniques and Tools*, 62–77. https://doi.org/10.1007/3-540-46020-9_5
[23] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (Nov. 1995), 39–41.
[24] Fiona Nah. 2003. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? *Behaviour & Information Technology - Behaviour & IT* 23, 285. https://doi.org/10.1080/01449290410001669914
[25] Michele Nitti, Virginia Pilloni, Giuseppe Colistra, and Luigi Atzori. 2015. The Virtual Object as a Major Element of the Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials* 18 (11 2015), 1–1. https://doi.org/10.1109/COMST.2015.2498304
[26] C. Perera, P. P. Jayaraman, A. Zaslavsky, P. Christen, and D. Georgakopoulos. 2014. MOSDEN: An Internet of Things Middleware for Resource Constrained Mobile Devices. In *2014 47th Hawaii Int'l Conf. on System Sciences*. 1053–1062.
[27] Juan Ramos. 2003. Using TF-IDF to determine word relevance in document queries. (01 2003).
[28] Thomas Tullis and William Albert. 2008. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics.*
[29] Blase Ur, Melwyn Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *CHI 2016*. 3227–3231. https://doi.org/10.1145/2858036.2858556
[30] WordNet [Online] 2019. wordnet.princeton.edu/.
[31] XMLTV File Format [Online] 2019. wiki.xmltv.org/index.php/XMLTVFormat.
[32] Jaeseok Yun, Il-Yeup Ahn, Sungchan Choi, and Jaeho Kim. 2016. TTEO (Things Talk to Each Other): Programming smart spaces based on IoT systems. *Sensors* 16 (04 2016), 467. https://doi.org/10.3390/s16040467