MASON: an Open Development Contextual Sensing Framework Enabling Reactive Applications

Nathaniel Wendt and Christine Julien
The University of Texas at Austin
{nathanielwendt, c.julien}@utexas.edu

ABSTRACT

Mobile devices continue to push the limits of contextually aware application intelligence. However, due to the complexity of contextual processing and programming, a centralized system that handles all mobile context processing is difficult to realize. The problem of defining such a contextual reasoning unit that uses an all-encompassing contextual ontology for all possible uses of context is not feasible nor useful. Furthermore, implementing custom contextual logic *ad hoc* per application is difficult due to the complexity of sensor monitoring and contextual reasoning and may be redundant across applications. In this work we propose an openly developed dynamic ontology formation that allows developers to contribute logical pieces to a greater network of contextual reasoning for use by application developers. Specifically, we introduce MASON, a framework for supporting modular contextual reasoning development by handling low-level sensor routing and abstracting data sources as composable and functionally reactive data streams. This provisions for high levels of abstraction for contextual logic developers that contribute to the framework as well as application developers that use it. We demonstrate the simplicity of developing with MASON and show, through an audit of open source applications, the increased contextual functionality offered, better enabling the next generation of contextually reactive applications.

1. INTRODUCTION

Mobile devices have become increasingly intelligent in catering to user needs. However, a large obstacle to further improving mobile applications lies in effectively leveraging available on-device sensors in order to reason about a user's context and to allow applications to make actionable decisions that better customize the mobile experience to the user. Part of customizing the mobile experience includes making proactive decisions on behalf of the user without explicit intervention. This means that mobile devices will need to offer more functionality but remain unobtrusive as envisioned by Weiser in his seminal paper on pervasive computing [36]. Currently, most applications simply respond to user input, making them largely passive. Rarely does a device automatically detect something about its own state and use that state to take action on behalf of the user, for the benefit of the user. This is due, in large part, to the *ad hoc* nature of current sensor sampling in mobile applications as well as the code complexity associated with reasoning about collected sensor data. Furthermore, with the exception of a few domain specific aggregations such as Apple's HealthKit[1], context derivation is not commonly shared across applications in a way that enables many applications to sample and reason about context efficiently.

We propose a new model in which a device constantly samples sensor data and performs system-wide contextual reasoning that can be shared across applications. This centralization of sampling and reasoning allows applications to subscribe to contextual updates, inducing application-specific actions behind-the-scenes or prompting the user for interaction. This enables a new breed of reactively intelligent applications that automatically respond and adapt to user context. To motivate this model, consider the following use case:

> *Greg grabs his mobile device as he leaves his apartment for his evening run. His device constantly monitors its onboard sensors and derives relevant contextual updates to which his fitness and music application are subscribed. As Greg begins jogging, this on-device contextual monitor determines that his movement profile has changed, causing a contextual update that spurs his music application to launch his favorite running playlist without any interaction from Greg. Once Greg completes his run, contextual updates trigger the music application to stop and the fitness application to query recorded spatiotemporal histories, launch to the foreground, and display a summary of his route and pace.*

While these capabilities are readily implementable on mobile devices today, the code required to implement them requires applications to respond to asynchronous sensor updates across many user contexts, which results in a very complex programming process. Furthermore, similar functionality would need to be redundantly programmed in each of the fitness and running applications. In order to support simplifying this programming process for the user, a centralized contextual reasoning system is required that can process context and inform applications. Prior work has been

[1] https://developer.apple.com/healthkit/

MobileSoft'16, May 16-17, 2016, Austin, TX, USA
 ISBN 978-1-4503-4178-3/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897073.2897099

done on similar systems [9, 14, 17, 25] but all of the ontologies supported by these contextual reasoning frameworks are statically defined and do not encompass many contextual attributes that may be useful. We posit that creating an all-encompassing static ontology at deployment time is infeasible and not necessary for most application usages. We propose a framework, MASON, that facilitates open development of modular contextual components for which application developers can subscribe in order to tune applications to automatically respond to contextual updates as reasoned from raw sensor data. MASON introduces two key contributions: dynamic ontology support and reactive programming abstraction.

The first contribution of MASON is the formation of an ontology, or a semantic contract through which applications agree on interpretations of context. Creating this ontology is a significant design challenge that has been partially addressed in prior work, for example in the OWL based mobile ontology [34]. However, many challenges still remain in adopting a static globally useful knowledge mapping [8]. Rather than tackling the challenge of creating a fully encompassing ontology before deployment, we propose facilitating a dynamic ontology that is sourced from multiple application developers. We leverage the AWARE framework [11] to manage sensor sampling within our contextual engine, on top of which we provision for custom developed components, called contextual abstractions (CAb), that applications construct. In union, the available CAbs provide an ontology that is formed as required from application subscriptions. For the purposes of this paper, we include several example CAbs with our sensing engine, but additional CAbs can be constructed by any developer and packaged as standalone Android applications. We also implement a simple dependency management system that enables users to download CAbs on-demand as required by applications. This dynamic ontology allows applications to (implicitly) agree on context semantics without requiring a static outline of the semantics at framework deployment time.

The second primary contribution of MASON is a suitable level of abstraction such that developers avoid programming low-level sensor management code with complex callback chains. Through MASON, application and CAb developers simply indicate the component interests and desired accuracies and define the processing of the resulting updates. All communication, including component installation and registration, is handled by the MASON Library, encouraging the client programmers to focus on the logical processing of data updates. While providing abstraction is useful, it has been generally realized in the frameworks mentioned previously.

To further ease the burden of client development, we adopt the reactive programming paradigm (also known as compositional event systems) through Reactive Extensions [7], a library that simplifies composing asynchronous event-based programs through observable sequences. Essentially, in Reactive Extensions, both sampling of sensors and logical interpretations and compositions of sensed values are treated as continuous streams of data. Components can subscribe to or observe various streams or create their own compositions of streams using a variety of operators. Through this paradigm, we create a very high-level of abstraction and code simplicity to allow client developers to focus on the logic behind the reactive nature of the application rather than the implementation details related to sensing and sampling. A further boon to abstracting raw sensor data is that it can help preserve potentially sensitive raw data that applications may not even need anyway, further supporting user adoption.

The rest of the paper is organized as follows. We review motivating related work in Section 2 then present the novel MASON framework components in 3. Section 4 describe the framework's architecture and our prototype implementation of it. In Section 5 we present the API that developers use to access MASON's features and benefits. In Section 6, we audit several open source applications for potential uses of MASON and discuss concrete application examples. We conclude with a discussion of future development aims in Section 7 and a summary of the paper in Section 8.

2. RELATED WORK

The usefulness of providing contextual reasoning has been demonstrated across many applications. These applications include contextual reasoning such as providing insight into socio-economic factors from movements patterns [20] and cell usage [13], early warning signs of bipolar disorder [28], and physical activity [33]. Health-centric applications also demonstrate great potential for using contextual information from motion and audio sensing [29] and location and communication sensing [22]. Additionally, reactive applications demonstrate capabilities for acting on sensed health concerns in preprogrammed ways [19, 21, 24]. We borrow inspiration from these approaches to provide a framework that allows for all types of applications to be highly customized towards reacting to contextual information.

Crucial to the effectiveness of contextual reasoning is the widely explored area of mobile sensing. Maintaining a strong degree of energy efficiency is crucial to sensing in mobile environments and many approaches have been proposed for doing so. EmotionSense allows for declarative programming to improve the power saving of sensing [31]. Other approaches such as CenceMe [23] and SociableSense [30] explore offloading computations, but they require significant developer effort to partition the workload appropriately. We look to simplify the developer's task as much as possible in providing reasoning from mobile sensing. Orchestrator [18] offers a resource orchestration framework that generates logical and physical plans to determine the best sensing outcome. In [16], the authors demonstrate a programming sensing flow where developers register application level requirements like monitoring intervals and tolerable delays across sensors and the system optimizes the overall sensing task. Seemon [17] introduces high level context monitoring queries (CMQs) that allow high-level applications to subscribe to contextual updates from sensor values. We borrow two key ideas from Seemon: only updating context subscriptions on value changes or updates and implementing bi-directional control flow to allow the system to reflect on sensing requirements and monitoring requests to better support energy savings. In [35], the authors develop a hierarchy of sensors with respect to energy consumption and optimize sensing based on lower level sensors being used in place of more expensive upper level sensors. Our framework's dynamic ontology based on CAbs reflects a similar hierarchy to allow for information reuse to better support energy savings. Lastly, ACE [25] creates a system of contexters such as (isHome, isDriving, etc.) to which applications can subscribe. ACE uses both *inference caching* and *speculative sensing*; the former infers one

contexter attribute from another without acquiring sensor data and the latter infers the value of an expensive attribute by sensing a cheaper one to improve energy savings. The CAbs in our system are similar to contexters in ACE, with the exception that they are generally more extensive, with each CAb potentially providing several contextual states. It is important to note that many of these energy saving approaches are complementary to MASON or could be generally applied as a sensor management scheme. Our focus is not on the energy savings of mobile sensing, but rather on provisioning for a dynamic ontology that supports asynchronous contextual events at a high level of abstraction for the application programmer.

The framework we propose is motivated by the seminal work of a conceptual framework for conceptual processing proposed in [9]. Our key motivations stems from the aim to create abstractions that encapsulate common context that support applications. The CORTEX project provides a model of sentient objects for developing context-aware applications in *ad hoc* wireless environments [32]. This model treats sensors as producers of streams of events and software components as consumers of the stream, much like the reactive paradigm that we adopt. Another similar work, Open Data Kit [14], creates a framework for reusable sensor drivers for external sensors to connect to mobile devices. Integrating new sensors is possible by downloading capabilities from the application market without modifying the system. While we deal with on-device sensors, we share a common aim to provide reusable contextual reasoning components that can by dynamically added to the system to support high level application uses.

The framework we propose also touches on the field anticipatory computing to better tune applications to user access behavior by relying on past, present, and anticipated future in order to make actionable decisions [27]. Google Now [6] implements end-user tailoring through contextual monitoring to supply anticipatory computing. Our framework does not operate at the browser level, but aims to allow applications to dynamically tune to user context in an anticipatory manner. Previous studies investigate ideal conditions to deliver notifications to prevent user interruption such as between detected user state activities [15], after completion of an event such as sending a text message [12], or across various user contexts [26]. Ultimately MASON goes beyond limiting interruptibility of notifications by providing application developers with the tools necessary to tune the responsive and anticipatory nature of their applications to the appropriate contexts.

3. FRAMEWORK

The key components in MASON are the logical contextual processing units that form contextual abstractions, or CAbs. Developers and domain experts design CAbs that process inputs from various sources such as sensor measurements, cloud queries, and other CAbs to provide high levels of contextual reasoning and abstraction. CAbs might include anything from determining user physical activity to user emotional health. Applications then use the outputs of CAbs to easily create contextually aware applications that do not require tedious sensor management or input processing. Multiple applications can use the same CAbs, reducing the amount of redundant processing and logic required.

3.1 CAb Development

A key contribution of MASON is providing a programming interface that greatly reduces the overhead required to handle asynchronous updates formed from sensor sampling. MASON allows developers to focus on the *logic* of processing various inputs rather than the *setup* and *management* of these inputs. We now discuss the process of implementing a custom CAb with the MASON Library as illustrated in Fig 1.

3.1.1 Naming

Uniquely identifying each CAb is essential for MASON to determine dependencies, appropriately route updates, and request user installations if necessary. MASON abstracts the communication requirements of a CAb developer by only requiring the implementation of two methods for identification: `getDisplayName()`, which allows for clients of the CAb to identify the human-readable name of the CAb and can help with debugging, and `getId()`, which determines a unique identifier for the CAb. CAbs and applications use this identifier to subscribe to the CAb. With this naming in place, no communication code is required by the CAb developer to receive registrations. All of this communication is handled internally within MASON and the MASON libraries.

```
public class Safety extends Cabs {
    @Override
    public void init() {
        MasonMediator med = new MasonMediator();
        Observable.combineLatest(med.gps(0.7),
            med.cab(AbstractLocation.ID, 1.0), (x,
            y) -> process(x, y)).subscribe();
    }

    public class Data {
        public int value;
        public Data(int value){ this.value =
            value; }
    }

    @Override
    public String getDisplayName() {
        return ‘‘Example’’;
    }

    @Override
    public String getId() {
        return ‘‘com.ut.mpc.cabs.example’’;
    }

    public Sample process(Sample gpsSample, Sample
         absLocSample){
       // business processing here
       onNext(new Data(0.75), false);
    }
}
```

Figure 1: Example CAb outline with required methods.

3.1.2 Schema

A CAb developer defines the schema that the CAb will use for update values by declaring an inner class named `Data`, indicating the desired structure. This provides implicit documentation for clients of the CAb (applications or other CAbs). Encouraging proper communication between developers is essential in the open source and dynamic nature of CAb installation and use. Inner class schemas are packaged for transmission by transforming class members to

a JSON formatted string by use of GSON[2] and wrapped in a `Sample` or `ContextSample` class.

3.1.3 Registration

It is essential that registering for sensor and CAb updates is simplistic. As shown in Figure 1, the `init()` method, invoked upon CAb startup, contains the registrations to external components as well as the functional operators performed locally on these components. The mediator object handles all communication with the context engine and with other CAbs. Methods invoked on the mediator object indicate registrations for various CAb and sensor components. The mediator handles checking for the existence of CAbs, requesting the user to download missing CAbs, and registering for sensor updates while presenting the reactive programming paradigm for the client developer. CAb developers program as if the observable streams are locally available, greatly simplifying the processing of remote CAbs and sensor data. Furthermore, double values are passed to mediator method invocations to indicate accuracy requirements for the given component and are passed along with the appropriate registration. CAbs can resolve the required accuracies from subscribed components and can adjust methods for computation (e.g., whether or not to offload computation) in the event of high accuracy requirements.

In the example `Safety` CAb in Figure 1, the GPS sensor updates are merged with the `AbstractLocation` CAb updates and the `process()` method is invoked when either stream updates. Separating the registration in the `init()` method from the sample processing is a design decision to encourage developers to separate data processing from stream processing and component registration. For simplicity, we have included the process method within `combineLatest()`, but it is best practice to include this method within the `subscribe()` method, at the expense of a separate combine function and process function.

3.1.4 Updates

The primary function of CAbs is to provide contextual output updates to other CAbs and applications. The `onNext()` method is the only method that the CAb developer must call to forward a contextual sample. The `Data` object discussed previously is passed to the `onNext()` method and contains the content of the CAb update. All components that have registered with the CAb are forwarded the corresponding data sample without any effort from the CAb developer. Note that no client code is required to handle incoming component registrations for the CAb, as this is also handled behind the scenes. The second parameter to the `onNext()` method indicates whether consecutively repeated values should be emitted.

CAbs are packaged as standalone Android applications. CAbs developers have the option of only implementing the standard CAb interface that will operate in the Android system background but can also include a user interface component to help provide feedback and training to any machine learning models present within the CAbs. An example is requesting the user to help train with activity recognition or familiar location detection such as home,work, etc.

3.2 Dynamic Ontology

Another key contribution of Mason is that application requirements drive the dynamic formation of CAbs. Applications register for CAbs by their unique identifier and indicate computation accuracy requirements. Mason checks existing CAb installations and requests the user to install any missing CAb dependencies, a process which could also, optionally, be automated. CAbs are composable and may depend on other CAbs, forming a dependency hierarchy that also enables CAbs to reuse computations provided by other CAbs. Mason performs a depth-first traversal of this dependency tree in order to resolve all dependencies. Mason then begins sampling from data sources only as required from the CAb dependency hierarchy. The result is the minimum sampling required to provide the subset of all context that is required by the applications on a given device. This customized contextual specification, provided by the union of CAbs, forms a dynamic ontology that is unique to each device and user's application requirements. This flexibility relieves Mason from forming some all-encompassing static ontology that enumerates all possible current and future application needs. Developers provide contextual abstractions through CAbs and context features, states, and specification contracts can be added and removed to form a *minimum dynamic ontology* per-device. Adding and removing CAbs can occur at any time as required by applications, allowing this dynamic ontology to reform over time to meet the application-level requirements.

3.3 CAb Hierarchy

To illustrate an instance of the dynamic ontology created from a hierarchy of CAbs, we have developed several examples CAbs. As shown in Figure 2, the CAbs conceptually reside above sensor data sources and form a hierarchy of inter-CAb dependencies. The physical sensors that we use for these CAbs are GPS, Accelerometer, and Bluetooth. We also wrap the Android system calls to application states (foreground, background, crashes) and communication calls and messages (incoming, outgoing) to create an updating stream modeled as a sensor. Therefore, when we speak of sensors, we speak generally as a source of information that can be modeled as a stream. We have also included a spatiotemporally indexed history of location and timestamp data, labeled as ST DB. This component is a database that is populated from GPS measurements and is not treated as a sensor stream, but rather, as a database element that CAbs may query. Keep in mind that these data sources were chosen for example purposes only, and any number of sensors and sources can be included in our framework for use with CAbs.

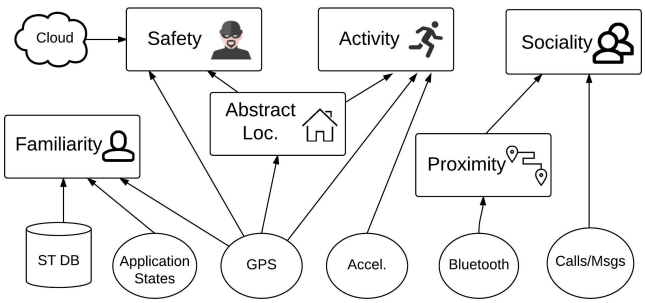

Figure 2: Example CAb Hierarchy

[2]https://github.com/google/gson

CAb	Values
Familiarity	0.0 - 1.0
Abstract Location	Home, Work, School, Unknown
Activity	Still, Walking, Driving, Unknown
Proximity	Family, Friends, Coworkers
Safety	Safe, MidSafe, Unsafe, Unknown
Sociality	0.0 - 1.0

Table 1: CAbs and Values

We created six CAbs:

- *Familiarity* monitors spatiotemporal histories (location traces) as well as application states and current GPS coordinates to determine how familiar a user is with a context across space, time, and device usage.

- *Abstract Location* translates raw GPS data into more abstracted locations such as work or home. This CAb is a good candidate for connecting to a UI through which the user can train the CAb by tagging raw locations with these abstract locations.

- *Safety* assesses the user safety by performing an HTTP request to an API that provides crime statistics for a location as well as checking Abstract Location updates. By monitoring the Abstract Location updates, this CAb can reuse computations and prevent further processing. For example, an Abstract Location value of home may have a fixed safety value without having to perform an HTTP request.

- *Activity* computes the physical activity of the user by monitoring Abstract Location updates as well as GPS data and accelerometer measurements. Much like Safety, Activity can avoid unnecessary accelerometer or GPS processing by receiving Abstract Location updates that infer a physical activity. For example, if the user is at home, they are likely either still or walking.

- *Proximity* monitors Bluetooth sensor readings to detect nearby Bluetooth enabled devices. Bluetooth device IDs are then mapped to known entities such as friends or family.

- *Sociality* captures the degree to which a user is being social by synthesizing Proximity CAb updates and device calls and messages.

The possible states of these CAbs are given in Table 1. CAb updates can take enumerated values as well as integer values, as chosen by the CAb developer. In some cases, applications may need raw sensor values such as with GPS coordinates. Rather than allowing applications the ability to register for sensor stream updates as CAbs can, we encourage developers to create a custom CAb that processes and creates updates containing the required data. Recall that application developers can include their own CAbs within their application project. This preserves our dependency management model and also maintains re-usability of computations that may be performed on the raw sensor data before sharing with applications.

Operator	Function
Buffer	group items emitted by an Observable
Filter	emits items only that satisfy a supplied predicate test
CombineLatest	applies a function to multiple Observables when any of them emit an item
Scan	apply a function sequentially to each emitted Observable

Table 2: Useful ReactiveX Operators

3.4 Reactive Extensions

To address the complexity of asynchronous events such as sampling sensors, we adopt the ReactiveX API, specifically the RxJava implementation[3]. ReactiveX combines elements of the *Observer* and *Iterator* patterns while providing functional operators for easily processing streams of data. We choose to treat both sensor samples as well as contextual updates generated by CAbs as streams of data. In reactive programming terminology, these streams are treated as *Observables*, or sequences of emitted data items, and components can subscribe to the *Observable* to receive data values. Subscriptions require a function to invoke upon an *Observable* emitting a new value, and can be conceptually viewed as a callback. *Observables* differ from the standard callback paradigm because they are composable, allowing for combining, filtering, and mapping in addition to several other operators. By using ReactiveX and *Observables*, the implementation of our contextual engine, as well as developers using the framework, can avoid complex asynchronous programming, enabling great programming power with few lines of code.

Our proposed use of *Observables* is not intended to serve as a data store or record such that identical values are repeatedly outputted, but rather, only when a change in state is detected. For example, the Activity CAb would not emit two consecutive values of "Driving". This allows other CAbs and applications to be designed so action can be taken in their subscription such as launching a new activity without worrying about repetition such as launching the new activity repeatedly. However, there may be cases where a CAb developer may to design a CAb that emits repeated items such as when Activity's value of "Driving" does not change but the associated accuracy does. In order to support this developer freedom, our framework supports both designs.

To illustrate the composable nature of ReactiveX *Observables*, we outline several useful operators in Table 2[4]. For use cases involving these operators see Sec 6.

4. ARCHITECTURE

In order to support CAb development and operation, MASON must handle multiple sensor integrations, CAb registrations and dependency resolution, and data stream routing. We now discuss how MASON performs these duties as framed under the overall architecture as shown in Figure 3. The figure includes the example CAbs we have developed to indicate where the CAb hierarchy forms within the framework.

Resolving CAb dependencies is crucial to supporting the

[3]https://github.com/ReactiveX/RxJava

[4]A complete list of operators can be found at: http://reactivex.io/documentation/operators.html

open nature of CAb development. MASON facilitates this process by requiring CAbs to register with their unique identifier upon initialization, storing these subscriptions in a table for lookup. Applications and CAbs that require other CAbs first check with MASON for availability through the CAb discovery component. Requests for CAbs that have not registered, and thus are not contained in the subscription table, prompt a user notification to download the CAb. Once the appropriate CAb is installed, it registers itself with MASON and the availability request completes, indicating the client of the CAb can proceed.

It is important for MASON to abstract away tedious sensor initialization and management code. To support this, CAb registration is simplified to only include the required sensors and the desired sampling accuracy for each sensor. Recall from the previous section that the CAb developer does not need to explicitly perform this registration and communication as it is handled by the MASON Library. In order to monitor sensors, MASON leverages AWARE[5] to interface with Android physical sensors and data sources. AWARE distributes commands to begin sensing to each data source and allows for tunability in frequency and accuracy of sampling. Sensor receivers within MASON register for updates from AWARE; these updates are triggered anytime the sensor generates a new data value.

The Sensing Logic Unit within MASON, driven by CAb registrations, determines which data sources should be monitored and when to activate them (and ultimately which sensor receivers to activate). This allows for MASON to support a wide array of sensor sources but only sample the sources as required from application requirements, realizing the lowest sensor level of the *minimum dynamic ontology*. While the main focus of this work is on developing a framework for the dynamic context ontology, future work optimizations could be performed in the Sensing Logic Unit to determine a sampling frequency plan that preserves the most energy efficiency. MASON is compatible with several prior works discussed in Section 2 that focus on this optimization of sensor sampling plans.

Once a sensor receiver receives a sensor source data sample, all subscriptions are checked with the corresponding sample. Since samples will only be received for which subscriptions exist, it is guaranteed that at least one CAb will be notified. MASON then forwards the received data sample to all CAbs that hold a subscription for that sensor source. CAbs will in turn process the data values and will likely output update values that are sent to other CAbs and context-aware applications.

4.1 CAb structure

We have previously discussed how CAb developers create custom CAbs for use with MASON without concern for registration and communication details. Now we outline how the MASON Library performs registration and updates. The generic internal architecture of a CAb is shown in Figure 4.

The registration component receives incoming registration requests from other CAbs or applications and stores these subscriptions. Incoming registrations to CAbs include a required certainty or accuracy from the CAb. CAb accuracy is defined by the CAb developer and the significance of values may vary widely between CAbs. The CAb developer may choose to implement tunable logic according to this accuracy

[5]http://www.awareframework.com/

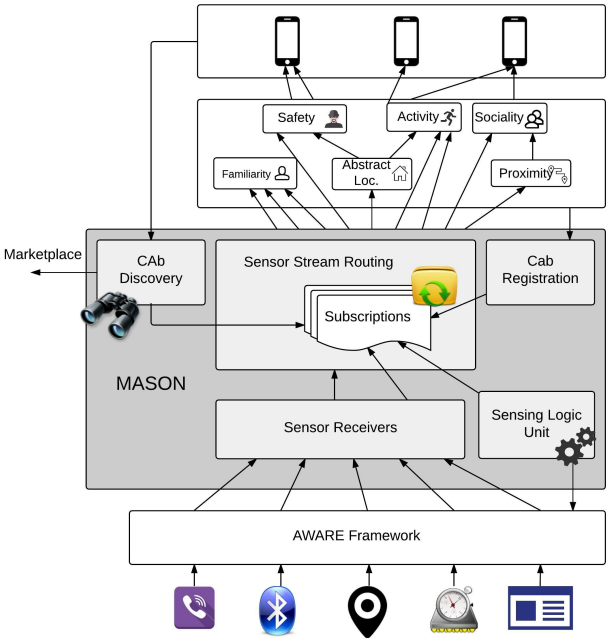

Figure 3: System architecture

such as offloading computations if the required accuracy is above a certain threshold. A well-documented CAb should document the effects of accuracy requests on the resulting contextual updates. The first incoming registration activates the CAb and spawns a registration with MASON and with any other CAbs. A CAb may also choose to synthesize incoming accuracy requests to tune outgoing registration accuracy requirements.

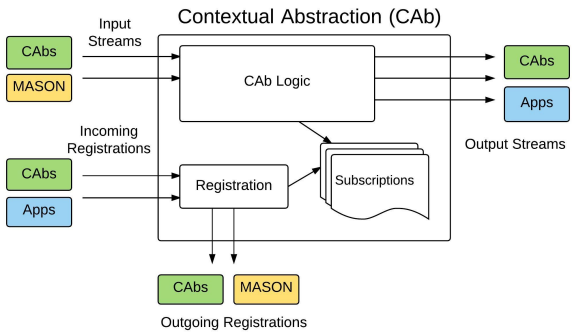

Figure 4: CAb architecture

Data inputs are received from other CAbs or sensor sources routed from MASON. The CAb then performs any logical computations on the inputs to determine the new contextual state value(s) from the new inputs. CAbs may be designed to repeatedly emit equivalent updates when an input changes or they may be designed to only emit updates when the contextual state has changed. Additionally, there is no restriction for coupling the timing of inputs and outputs, so the CAb developer has freedom to determine any periodic stream of contextual outputs. For example, a CAb may be created that generates fixed interval outputs regardless of input value timings. When the CAb determines that an up-

date should be generated, the subscriptions table is queried and appropriate CAbs and applications are updated with the contextual state values. Note the interconnected nature of CAbs as each CAb may interface with other CAbs across input and output streams as well as incoming and outgoing registrations.

4.2 Implementation

MASON performs all communication between components through public Android broadcasts. All components that receive messages listen for broadcasts by instantiating Broadcast Receivers. Most messages are sent as standard Broadcasts with the exception of queries to the ST DB and CAb existence requests to MASON, which use Ordered Broadcasts to make use of message responses. Public Android broadcasts can be a security concern [10], but mitigating these security risks is left to future work.

All CAbs extend from a base CAb class that is an extension of the Android Service class. Therefore, all CAbs operate as running background services to maintain data state, but adaptation of MASON Libraries is possible such that CAb state is maintained in databases that preserve state. Implementing the non-service approach is a straightforward extension that is left to future work for CAb developer convenience.

In order to fully realize the abstraction and simplicity of reactive functional programming, MASON supports lambdas by compiling the project with Java 8 and using Retrolambda[6] to support execution on the Android runtime. Code segments included in this paper use this lambda syntax, reflecting the actual implementation of MASON. This project is open source and open for contribution[7].

5. API

Similar to CAb programming, the process of including MASON into a context-aware Android application relies on high levels of contextual abstraction. The application developer never handles sensor level code directly and instead only declares high-level CAb requirements. There are two methods for implementing context-aware applications through MASON: Sentinel services and Reactivities.

Sentinel services are Android Services that passively monitor CAb updates and then subsequently pro-actively launch appropriate Activities. The Sentinel is meant to launch activities when the encompassing application is not in the foreground. The Sentinel may change the foreground activity for an active device or may wake the device entirely, with appropriate Android wake locks in place.

Reactivities are extensions of the Android Activity class and exist to monitor CAb updates and change aspects of the current foreground Activity. Reactivities may transform the UI, perform network requests, or any other function useful for tuning the application to the user context. The monitoring performed in Reactivities will not occur when the Activity is not in the foreground, any desired background monitoring should be done in Sentinels. Note that both Sentinels and Reactivities may launch new Activities, the difference lies in whether or not the monitoring is done in the foreground or background.

The process of creating a Reactivity is similar to that of programming a CAb. An example activity, `ChatReActivity`, is given in Figure 5, with the corresponding data flow modeled as streams in Figure 6. Note the familiar `getId()` and `getDisplayName()` methods as well as the familiar `mediator` object for handling communication and registration. As demonstrated in `ChatReActivity`'s `init()` method, the application subscribes to the Activity CAb updates and invokes the `process` function on received samples. `ContextActivity` subscribes for *Sociality* CAb updates and evaluates the average across 5 values. If the average is greater than 0.8, the `Social` Activity is brought to the foreground. Note the inclusion of the `submit()` function on the `mediator` object since the activity needs some action to start the registration and subscription process. Creating a Sentinel is similar to CAb and Reactivity development, with the same methods required for development. The only additional requirement is that Sentinels must extend the `MasonSentinel` class and be launched for initialization by the application programmer. An example Sentinel is given in Section 6.

```
public class ChatReActivity extends MasonActivity
{
    @Override
    public void init() {
        MasonMediator med = new MasonMediator();
        med.cab(Sociality.ID, 0.8)
        .buffer(5)
        .subscribe(sample -> process(sample));
        med.submit();
    }

    @Override
    public String getDisplayName() {
        return ‘‘Example’’;
    }

    @Override
    public String getId() {
        return ‘‘com.ut.mpc.cabs.example’’;
    }

    public Sample process(List<Sample> socSamples){
       if(average(socSamples) > 0.8){
          Intent intent = new Intent(this,
              Social.class);
          startActivity(intent);
       }
    }
}
```

Figure 5: Example Reactivity outline with required methods.

This example demonstrates the ease with which an application developer can implement high level context in an application without complicated callback code. Much like CAb development, the developer need not be concerned with sampling frequency and sensor management, but rather with designing the logical reactive components of the application.

6. CASE STUDY

In addition to bringing high level contextual abstractions to developers, it is crucial that MASON offers practicality and is useful in real world applications. As a means of evaluating these aspects of MASON, we perform an audit of several open source applications and indicate potential integrations of MASON. Note that this audit is not intended to be exhaustive as there are many more integration possibilities, but rather to motivate real world uses of MASON. Also, we

[6]https://github.com/evant/gradle-retrolambda
[7]https://github.com/nathanielwendt/LSTAndroid

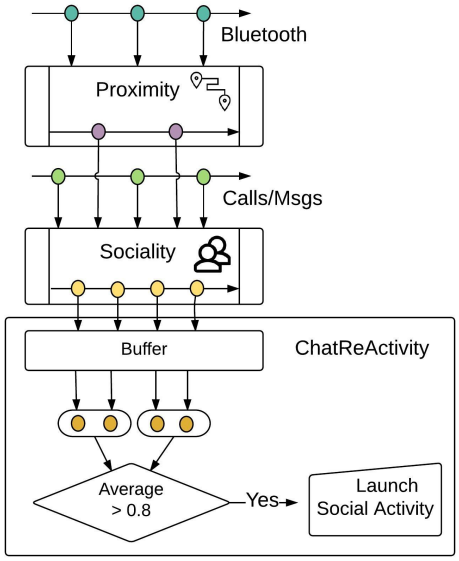

Figure 6: ChatReActivity Data Flow

only outline use cases leveraging the example CAbs that we have developed. There are many more CAb development possibilities that further increase the potential uses of MASON.

We selected five open source Android applications that varied across code complexity, from one activity to complex controllers and game mechanics. The applications also varied across domains, including leisure applications to security-intensive applications. The five applications are:

- *FotoFinder*[8] – photo viewing and management application
- *BankDroid*[9] – banking application for Swedish banks
- *Apollo*[10] – music player application
- *AndroidRun*[11] – physical fitness application for displaying distance and pace
- *AndorsTrail*[12] – single-player fantasy role playing game

Recall from the previous section that application developers can choose to implement MASON through Sentinels or Reactivities. Possible Sentinel integrations of MASON within the case study applications are shown in Table 3 including potential activities that might be launched as well as the CAbs responsible for causing the action. Through these potential uses of MASON, we demonstrate the potential for applications that monitor context and pro-actively launch or adapt to changing context. Possible Reactivity integrations of MASON are shown in Table 4 including the activity that could be extended as well as the CAbs that could be monitored. These examples motivate the simplicity with which developers could adapt existing codebases to provide applications that offer enhanced contextual awareness. Many of these examples, such as preventing a new lock pattern in BankDroid if the user is in an unsafe area, would require extensive code portions to setup and monitor if implemented without capabilities like those MASON provides. MASON provides developers with this functionality with only a few lines of code.

[8]https://github.com/k3b/androFotoFinder
[9]https://github.com/liato/android-bankdroid
[10]https://github.com/adneal/Apollo-CM
[11]http://sourceforge.net/p/androidrun/code/ci/master/tree/
[12]https://github.com/oskarwiksten/andors-trail

```
public class MainReactivity extends MasonActivity
{
   @Override
   public void init() {
      MasonMediator med = new MasonMediator();
      med.cab(Activity.ID, 1.0)
         .scan( (x,y) -> detectRunToWalk(x,y))
         .filter(act -> act.isType(``RunToWalk''))
         .subscribe(x -> showSummary());
      med.submit();
   }

   public Sample detectRunToWalk(Sample x, Sample
       y){
      String xVal = x.data().get(``value'');
      String yVal = y.data().get(``value'');

      if((``RUNNING'').equals(x) &&
          (``WALKING'').equals(y)){
         return new Sample(null, ``RunToWalk'',
             null);
      } else {
         return new Sample();
      }
   }
}
```

Figure 7: AndroidRun Application Reactivity

It is important to note that the choice between Sentinel or Reactivity depends on the anticipated user state. For example, the `MonsterEncounterActivity` that is a potential Reactivity in AndorsTrail could be implemented as a Sentinel if the developer desired monsters to be generated proactively and launched. Similarly, the `FotoGalleryActivity` examples could be implemented as Sentinels or Reactivities depending on whether the application is anticipated to be in the background or foreground. In some cases, the developer may include a Sentinel and Reactivity that have similar functionality but are performed both when the app is active or not active. Ultimately, these examples demonstrate the potential feature extension that real world applications could use without any considerable developer effort.

Next we demonstrate two concrete examples of integrating MASON as framed by our motivating scenario of Greg running and listening to music. We choose the *AndroidRun* and *Apollo* music applications for these examples.

`MainReactivity`, as shown in Figure 7, is a possible extension of *AndroidRun*'s `MainActivity`. This extension monitors the Activity CAb and applies the scan operator which compares the current data value and the previous value according to the simple detectRunToWalk function. We include the `detectRunToWalk` method to illustrate the simplicity with which a developer can compare data values, although we omit some class casting for brevity. A filter operator then checks if the new Sample created in `detectRunToWalk` is of the appropriate type, and if so, the `showSummary` method shows the user's pace and timing, summarizing the run. A buffer operator could also be used to ensure that the user maintains the running state across several updates

App	CAb(s)	Launch Activity	Launch Purpose
AndorsTrail	Proximity	`ConversationActivity`	user encounters friend that plays game, in-game character meeting
AndorsTrail	Activity	`LoadSaveActivity`	user motion pattern may indicate play stoppage play, game auto-saves
AndroidRun	Activity, ST DB	`MainActivity`	user changes from running to walking, query ST DB, show route and pace
Apollo	Activity	`AudioPlayerActivity`	user changes from walking to running, autoplay running playlist
FotoFinder	Proximity	`FotoGalleryActivity`	user becomes close to someone tagged in photos, show photos they share
FotoFinder	Familiarity	`FotoGalleryActivity`	user goes to unfamiliar place, show photos of new location from Internet

Table 3: Application Audit for Sentinel Services

App	Location	CAb(s)	Description
AndorsTrail	`DisplayWorldMapActivity`	Activity	implement movement feedback based on user movement profile
AndorsTrail	`MonsterEncounterActivity`	Abstract Location	adjust the difficulty and type of monsters found based on users abstracted location
AndroidRun	`MainActivity`	Activity	lock device screen when transition to running
Apollo	`SearchActivity`	Familiarity, Abstract Location	autocomplete music last played with same familiarity and known locations
Apollo	`AudioPlayerActivity`	Proximity, Sociality	a friend is detected nearby or a user is active socially, reduce volume or pause music
BankDroid	`LockablePreferenceActivity`	Safety, Familiarity	prevent setting new lock pattern in unsafe or unfamiliar areas
BankDroid	`SettingsActivity`	Safety, Familiarity	disable certain settings in unsafe or unfamiliar areas
BankDroid	`MainActivity`	Proximity	hide or abstract exact account balances and details if strangers are nearby
FotoFinder	`FotoGalleryActivity`	Proximity	prevent deleting pictures when friends/family nearby

Table 4: Application Audit for Reactivities

(provided that the Activity CAb emits consecutive duplicate values). A Sentinel could also be similarly implemented if this desired functionality was required when the application was not in the foreground. This example demonstrates the simplicity with which a developer can detect state changes from running to walking without requiring any sensor management or tedious callback chains.

```
public class ApolloSentinel extends MasonSentinel
{
   @Override
   public void init() {
      MasonMediator med = new MasonMediator();
      med.cab(Activity.ID)
         .scan( (x,y) -> detectWalkToRun(x,y))
         .filter(act -> act.isType(``WalkToRun''))
         .subscribe(x -> launchPlayer())
      med.submit();
   }

   public void launchPlayer() {
      Intent intent = new Intent(this,
          AudioPlayerActivity.class);
      intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
      intent.putExtra(``FLAG'',
          ``RunningPlaylist'');
      startActivity(intent);
   }
```

Figure 8: Apollo Music Application Sentinel

Figure 8 shows `ApolloSentinel`, which supports the functionality of automatically playing music when the user starts running. The `init` method is similar the one outlined in `MainReactivity`, with the exception of including `detectWalkToRun` since the music should be started when the user starts the running activity. We omit `detectWalkToRun` due to its simplicity and similarity to `detectRunToWalk`. Once it is determined the user is running, the Sentinel launches the music player activity with an extra string value that indicates the activity should start playing media from a running playlist. The *Apollo* app may require some user configuration to indicate which playlist to play while running, or alternatively, a history of running could be maintained that tracked music commonly played while the user was running. This Sentinel could also be implemented as a Reactivity if the desired functionality was only necessary while *Apollo* was in the foreground.

7. FUTURE WORK

Our initial implementation of MASON motivates exciting areas of extension and additional work. Future research could investigate resolving errors from CAbs determining incorrect contextual state. Beyond determining if a CAb's model is incorrect or if a sensor measurement is inaccurate, MASON could support a feedback system for applications to indicate that a CAb value was incorrect or unsatisfactory to the user. MASON might also facilitate some type of UI rollback to reset to previous device state in the event of a CAb error.

Additionally, since MASON allows developers to make proactive apps that often launch to the foreground, some kind of foreground request mediation may be useful. In the future, developers might be encouraged to launch new activities through a UI mediation component within MASON that resolves priorities and ensures that only a single application can launch at a time.

Future work might also investigate some kind of marketplace for CAbs similar to the Android marketplace. Currently, there is no enforcement of uniqueness in CAb IDs, potentially creating issues if multiple CAbs share the same ID. A centralized marketplace could ensure all IDs were unique as well as support CAb visibility such that developers might

not make conflicting or redundant types of CAbs such as multiple activity recognition CAbs. This marketplace could also support developer reviews to indicate how well the CAb worked and what kind of energy efficiency it typically maintained.

As previously discussed, future work with MASON could improve sensor sampling efficiency by implementing one or more prior works in sensor sampling efficiency. MASON could also batch samples from sensors such as accelerometers to reduce the overhead of routing updates to CAbs from each sensor reading. To further support device efficiency, MASON could be incorporated at a lower system level to reduce the runtime overhead of the large number of required global Android Broadcasts.

Lastly, future work could investigate developing additional CAbs. Examples include a CAb for mapping shake patterns of a device such that apps could launch when a user shook them a certain way. This CAb would require a UI and user training. Other future CAbs might include an audible ambiance detection as sensed from devices microphone, or a mood detection CAb as processed from other CAbs and various sensors.

8. CONCLUSIONS

Motivated by real-world applications, we introduced MASON, an openly developed dynamic ontology formation framework that allows developers to contribute logical pieces to a greater network of contextual reasoning for shared use by application developers. We demonstrated the functionally reactive programming interfaces for implementing contextual abstractions, or CAbs, and the similar API for application developers to integrate CAbs into applications. We also discussed the dependency resolution system as a part of MASON to manage CAb installations and prompt users to install new ones, if necessary. To demonstrate potential uses of CAbs, we introduced several example CAbs as motivated by real world application usages. We concluded with a case study of open source applications to motivate potential implementations of MASON and to demonstrate reactive functionality in application design. Ultimately, MASON enables a new form of ontology formation by open source developers and dynamic dependency resolution to provide high levels of programming abstraction to application developers in order to provide new levels of contextually intelligent and reactive applications.

9. REFERENCES

[1] Andofotofinder. https://github.com/k3b/androFotoFinder. Accessed: 2016-01-04.

[2] Andorstrail. https://github.com/oskarwiksten/andors-trail/tree/master/AndorsTrail/src/com/gpl/rpg/AndorsTrail/activity. Accessed: 2016-01-04.

[3] Androidrun. http://sourceforge.net/p/androidrun/code/ci/master/tree/. Accessed: 2016-01-04.

[4] Apollo. https://github.com/adneal/Apollo-CM. Accessed: 2016-01-04.

[5] Bankdroid. https://github.com/liato/android-bankdroid. Accessed: 2016-01-04.

[6] Google now. https://www.google.com/landing/now/. Accessed: 2016-01-04.

[7] Reactivex. http://reactivex.io/. Accessed: 2016-01-04.

[8] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

[9] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.

[10] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE security & privacy*, (1):50–57, 2009.

[11] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. Aware: mobile context instrumentation framework. *Frontiers in ICT*, 2:6, 2015.

[12] Joel E Fischer, Chris Greenhalgh, and Steve Benford. Investigating episodes of mobile phone activity as indicators of opportune moments to deliver notifications. In *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*, pages 181–190. ACM, 2011.

[13] Vanessa Frias-Martinez and Jesus Virseda. On the relationship between socio-economic factors and cell phone usage. In *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development*, pages 76–84. ACM, 2012.

[14] Carl Hartung, Adam Lerer, Yaw Anokwa, Clint Tseng, Waylon Brunette, and Gaetano Borriello. Open data kit: tools to build information services for developing regions. In *Proceedings of the 4th ACM/IEEE International Conference on Information and Communication Technologies and Development*, page 18. ACM, 2010.

[15] Joyce Ho and Stephen S Intille. Using context-aware computing to reduce the perceived burden of interruptions from mobile devices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 909–918. ACM, 2005.

[16] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. Symphoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 211–224. ACM, 2012.

[17] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2008.

[18] Seungwoo Kang, Youngki Lee, Chulhong Min, Younghyun Ju, Talwoo Park, Jmwon Lee, Yunseok Rhee, and Junehwa Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages

135–144. IEEE, 2010.

[19] Predrag Klasnja, Sunny Consolvo, David W McDonald, James A Landay, and Wanda Pratt. Using mobile & personal sensing technologies to support health behavior change in everyday life: lessons learned. In *AMIA Annual Symposium Proceedings*, volume 2009, page 338. American Medical Informatics Association, 2009.

[20] Neal Lathia, Daniele Quercia, and Jon Crowcroft. The hidden image of the city: sensing community well-being from urban mobility. In *Pervasive computing*, pages 91–98. Springer, 2012.

[21] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T Chittaranjan, Andrew T Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. Stresssense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 351–360. ACM, 2012.

[22] Anmol Madan, Manuel Cebrian, Sai Moturu, Katayoun Farrahi, et al. Sensing the" health state" of a community. *IEEE Pervasive Computing*, (4):36–45, 2012.

[23] Emiliano Miluzzo, Nicholas D Lane, Shane B Eisenman, and Andrew T Campbell. Cenceme–injecting sensing presence into social networking applications. In *Smart Sensing and Context*, pages 1–28. Springer, 2007.

[24] Margaret Morris and Farzin Guilak. Mobile heart health: project highlight. *Pervasive Computing, IEEE*, 8(2):57–61, 2009.

[25] Suman Nath. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 29–42. ACM, 2012.

[26] Veljko Pejovic and Mirco Musolesi. Interruptme: Designing intelligent prompting mechanisms for pervasive applications. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 897–908. ACM, 2014.

[27] Veljko Pejovic and Mirco Musolesi. Anticipatory mobile computing: A survey of the state of the art and research challenges. *ACM Computing Surveys (CSUR)*, 47(3):47, 2015.

[28] Alessandro Puiatti, Steven Mudda, Silvia Giordano, and Oscar Mayora. Smartphone-centred wearable sensors network for monitoring patients with bipolar disorder. In *Engineering in Medicine and Biology Society, EMBC, 2011 annual international conference of the IEEE*, pages 3644–3647. IEEE, 2011.

[29] Mashfiqui Rabbi, Shahid Ali, Tanzeem Choudhury, and Ethan Berke. Passive and in-situ assessment of mental and physical well-being using mobile sensors. In *Proceedings of the 13th international conference on Ubiquitous computing*, pages 385–394. ACM, 2011.

[30] Kiran K Rachuri, Cecilia Mascolo, Mirco Musolesi, and Peter J Rentfrow. Sociablesense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 73–84. ACM, 2011.

[31] Kiran K Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J Rentfrow, Chris Longworth, and Andrius Aucinas. Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, pages 281–290. ACM, 2010.

[32] Carl-Fredrik Sørensen, Maomao Wu, Thirunavukkarasu Sivaharan, Gordon S Blair, Paul Okanda, Adrian Friday, and Hector Duran-Limon. A context-aware middleware for applications in mobile ad hoc environments. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 107–110. ACM, 2004.

[33] Emmanuel Munguia Tapia, Stephen S Intille, William Haskell, Kent Larson, Julie Wright, Abby King, and Robert Friedman. Real-time recognition of physical activities and their intensities using wireless accelerometers and a heart rate monitor. In *Wearable Computers, 2007 11th IEEE International Symposium on*, pages 37–40. IEEE, 2007.

[34] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. Ieee, 2004.

[35] Yi Wang, Jialiu Lin, Murali Annavaram, Quinn A Jacobson, Jason Hong, Bhaskar Krishnamachari, and Norman Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 179–192. ACM, 2009.

[36] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.