UC San Diego
Technical Reports

Title
An Ontology-Driven Context Engine for the Internet of Things

Permalink
https://escholarship.org/uc/item/2kn5t9zg

Authors
Venkatesh, Jagannathan
Chan, Christine
Rosing, Tajana Simunic

Publication Date
2015-02-02

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

An Ontology-Driven Context Engine for the Internet of Things

Jagannathan Venkatesh, Christine Chan, and Tajana Simunic Rosing
University of California, San Diego

***Abstract*—The Internet of Things (IoT) refers to an environment of ubiquitous sensing and actuation, where all devices are connected to a distributed backend infrastructure. The main benefit of the IoT is the ability to use myriad sensor data, leveraged into high-level information about the entities in the system for reasoning and actuation in *context-aware applications*. Significant growth in sensor deployment has lead to unregulated and diverse information being fed back to the system at large. A formal specification, or *ontology*, for data use provides regulation to the system. In addition, IoT middleware is required for context-aware applications to operate in an environment with constantly changing data, sources, and context. In this paper, we present a context engine for IoT applications founded on an ontology that specifies and reasons on context information. We explore and build upon related work on IoT needs and ontological principles. Our infrastructure leverages context information for learning and processing a changing environment. Finally, we implement two applications: one to demonstrate machine learning from heterogeneous, intermittent sources, and another with an end-to-end implementation of user-driven actuation using the IoT backend. In the former, we produce an output stream of context information 60x more accurate than either of the individual sensor streams alone. The latter exemplifies the ease of development and extension, with only 20% infrastructure-related overhead.**

***Keywords*—Context aware computing; Internet of Things; Sensor ontology**

I. INTRODUCTION

Sensor networks and ubiquitous sensing are evolving into a new concept – the Internet of Things (IoT) – the collection of sensing and actuation backed by the existing and growing Internet infrastructure [1]. This creates a unique scenario from prior sensing approaches: the pre-IoT work in this area still envisioned a level of modularity and control over the sensors in the systems [2]. However, the practical reality is that the emerging implementation of the IoT has multiple distinct systems communicating with their own web-based backends, exposing distinct APIs for interaction and data retrieval. These heterogeneous devices are added, removed, or updated independently of each other by different manufacturers with different goals and release cycles, and the choices among them are entirely in the hands of the user. This precludes the unifying vision of the IoT from an academic perspective. The wearable fitness tracker market is a current example: Fitbit, Garmin, Moves, Microsoft, and Apple, the most prominent among several, have developed trackers and independent backends. Dozens of applications exist that independently scour the data stores for different pieces of relevant data to aggregate metrics for a user's goals and progress. Extension of applications to new devices and APIs is a manual process requiring a redeployment of the user-facing application itself [3]. If the data sources and types change, the backend of the application might require reimplementation as well. Existing sensing and context infrastructures [4], [5] implement strong ontologies but lack support for such a changing system, as applications need to constantly adapt to the environment and the constituent devices.

To address this, we focus on an application framework for these context-aware applications, which we call a *context engine*. In order to balance the IoT needs with ease of development on the application side, we leverage a context-focused ontology in our implementation. The goal of this work is a middleware framework that bridges the gap between the data in the infrastructure and the applications in the IoT. The context engine enables translation of heterogeneous sensor data into high-level, usable context, and allows applications to reasoning, optimizing, and processing based on dynamically available sensor data. We can achieve this while still enabling the functionality and scaling of IoT applications.

We provide an overview of the related work in IoT and sensor ontologies, and characteristics and requirements of context-aware applications in Section II, identifying from among them features that are appropriate to our goals. We then design the context engine system in Section III, leveraging specifications that help us meet a context infrastructure's needs. We outline the base context engine implementation in Section IV. Finally, in Section V, we design two independent applications: one that implements the system in an actual IoT infrastructure, and another that establishes the ability of this ontology to handle machine-learning and adaptation.

II. RELATED WORK

There is a large body of work on sensor ontologies, context awareness, and the Internet of Things. Perera et al. [1] provide a comprehensive overview of context-aware

computing, covering fifty publications over the last decade. They evaluate system behavior and application development in regards to their applicability for the IoT, using an IoT taxonomy based on the features and models identified in related work. The authors identify four interactions with context in an IoT infrastructure: acquisition, modeling, reasoning, and distribution. It also identifies several important design requirements for context awareness in the IoT: *hierarchical model*, scalability, context *life cycle management*, *extensibility*, *flexibility* for multiple reasoning modes, *resource sharing*, *optimization*, and *automatic event detection*. It uses these requirements to illustrate the gaps that comprehensive IoT middleware should fill and identify areas of difference from other ubiquitous sensing approaches. When designing our system, we make sure to consider all the features above.

A. Context Ontologies

Ontologies are formal data representations that categorize the vast amount of unregulated and diverse data from information sources in the IoT [6] . They help classify the data provided to applications, but the organization of the IoT still makes application deployment difficult.

There are many previous works [7], [8], [9] that outline formal context models for domain-specific designs, but do not intend to share data or actuation beyond their original one-off applications. However, pervasive sensing provides many of the ontologies that are now adapted to the Internet of Things. One of the earliest is the resource description framework (RDF) [6], which annotates all web objects with semantic information, implemented in XML, a web-ready format. The aggregation of these annotations forms a directed graph that is already used in context-aware web applications such as search engines. While this is adequate for objects such as websites, as was its original goals, this binary object-object connections are insufficient for the Internet of Things, which requires a richer description of relationships, as well as an easier way to query and determine these relationships, make inferences, etc.

To address these limitations, the successor to RDF was the Web Ontology Language (OWL), now a web standard, which also transitioned from loosely defined and typed systems suitable for Wireless Sensor Networks to a formal ontology [10]. OWL has been designed as a hierarchical system, with sub-domains of objects (e.g. appliances) encapsulated by the domains they live within (houses). Several systems have been designed using OWL: smart spaces [10], meeting room organization [10], hierarchical modeling of the physical environment [11]. OWL-S is an implementation built on top of OWL for describing semantic services [6], adopted by [12] with an emphasis on scalability and testability of their services model.

However, OWL typically operates under a very strictly defined hierarchy. Although this works well for static applications that rely solely on a fixed set of data, IoT applications may have to deal with changing sources and sinks. Nodes should be removable and the system should still be able to operate to the best of its capacity. In addition, the amount of data for which each application must be responsible can grow rapidly, as the amount of infrastructure-related data (dependencies, relation annotation, etc.) that each application needs to manage can grow faster than the data itself.

Another extension of OWL handles this issue: the Context Modeling Language (CML) [13]. The system is based on the Object-Role Model, which is preferable for the Internet of Things, as all context data is attributed to a physical or virtual entity (the object) and provides a particular form of information associated with it (role). The ontology reverts to a flatter hierarchy than OWL – this allows an application to deal with as little context as it requires, rather than be responsible for the entire graph. Furthermore, CML provides a direct web-oriented communication language in XCML, a markup implementation that is very important for an Internet-based backend. We borrow much of our syntax from XCML. This provides an ontology for mapping "names" of context variables to their "values". The *context space* includes all possible context variable names, which may grow according to any additional data types the ontology describes. Finally, to aid the adaptability of context-aware applications, CML introduces the concept of *context dimensions*, which define the minimum subsets of context spaces, in which an application can perform meaningful computations and operations. By specifying dimensions in the same markup as the data, CML opens the doors for the specification itself to be changed to adapt to the data sources available.

B. Context-aware Applications

Pervasive sensors gather raw data from the environment and its objects, which must be parsed into higher-level information. We refer to this high-level, filtered and processed information as *context*. High-level context provides a representation that is both lower in computational overhead and more intuitive for application developers to use in reaction to IoT sensor stimuli, such as crafting cognitive assistance for medical patients [9], or custom learning environments for online students [14]. These context-aware applications represent the major advantage of the Internet of Things [1].

The smart home application in [7] presents a good example of the diverse sources found in the IoT and used by the constituent applications. The system uses a combination of sensors, user-supplied information, and higher-level data processed from mobile and computing devices. However, it does not account for adding, removing, or changing the format of context in the nodes of the dependency tree.

Hong et al. [4] build context information based on a host of labeled environmental and user sensor data (e.g. biometrics, GPS location, interaction with phone, weather, etc.) and context rules. Lee et al. [15] present a location prediction model based on a dynamic Bayesian network, where accuracy significantly exceeded static networks. The location model enables their ultimate goal of supporting

ubiquitous computing decisions. With the growing popularity of alternative education, e-learning systems can personalize learning materials and recommendations based on modeling student profiles and context [5]. There are many more mobile applications that operate on context awareness for localized user information [14], vehicular safety [7], or battery saving [3].

While streaming data from human subjects is natural, sliding windows of the continuous data must be smoothed and preprocessed before inputting into an analytic or modeling framework. Some works further apply machine learning techniques to model user behavior and interaction with their physical environment. K-means clustering is a prevalent way to automatically relate low-level data into high-level contexts [4]. Reinforcement learning is an important learning method for context awareness in IoT applications, as users are already innately involved in sensing and actuation. It invites user interaction to reinforce and guide the system towards better accuracy and intuitive actuation. For example, Madhu et al. [9] schedule reminders for a user who is cognitively or orthotically impaired. They use temporal constraint reasoning to describe a daily plan and reinforcement learning (function approximation-based learning) to find optimal actions, subject to adjustable human parameters.

Lee et al. [8] remove reliance on labeled data by performing unsupervised learning over low-level sensor event sequences to extract patterns that represent high-level activities, even if the activities are discontinuous or varied over time.

C. *Context-Aware Middleware*

In an IoT context, middleware is software that acts as a bridge between low-level sensors and the backend, or between the backend and user-facing applications. Other works have explored context-aware middleware frameworks. Petrera et al. [16] propose a sensor configuration model for the IoT that can handle sensor filtration and reasoning, implemented via asking the user a series of questions. However, the reasoning seems to fall short of an open platform for machine learning, instead limiting reasoning to an annotated dependency graph. Similarly, [17] overviews several middleware implementations, identifying certain aspects of each: ontology-based, flexible for reasoning, data filtration and adaptation, but does not present a single solution encompassing all aspects.

D. *Takeaways*

The ontologies and applications in the related work help clarify the requirements common to context-aware applications in the Internet of Things:

1. **Adaptability:** The system allows reasoning under multiple data subsets
2. **Distributed Data Aggregation:** Data comes from various sources, must translated into a widely understandable structure and format
3. **Data Flexibility:** Internal and external data representations should be flexible enough to handle complex sensor data – continuous vs. discrete data, streaming and persistent data, and the more complex implementation of high-level context [1].
4. **Reasoning:** The context engine is to be developed into context-aware applications, and the ontology must reflect this. To maintain this, we allow an extensible framework internally, while enforcing only ontology-driven input/output.

However, an application-facing infrastructure that enables these properties is missing. The various proposed ontologies are only applied towards very specific applications [10], [4], [9], [8] – there is no larger system within which the applications exist, nor is there the ability for other applications to use the resources made available to the one, as should be in the real Internet of Things. The closest entity is the middleware layer implemented in [8], which provides a general framework, but only within the domain of "smart spaces", and again, specific to the application that is designed.

More importantly, none of the related works account for application adaptability. The emerging IoT implementations require the ability to merge data from different sources and backends into a common context representation, and on the application side, to seamlessly incorporate new data or formats without interrupting the application. Regardless of whether adaptation is a product of a changing environment or sources joining and leaving the sensed surface, the framework around the applications are made static. CML [13] was the only ontology to allow application processing using different sets of available data, but the implementation covered only the ontology and not the applications.

Real IoT applications need to exist in an infrastructure that fosters changing the data, sources, and internal reasoning. The context engine developed in this paper fits this gap in the current landscape. We present a framework and methodology that developers inherit, allowing them to describe their source data and end-user applications in terms of their context space and dimensions. This enables applications that are flexible with new sources and dimensions, and whose behavior can be extended, even during runtime.

III. System Design

We use the conclusions drawn from the previous section to design our infrastructure. Our goal is to enable the design of context-aware applications in the presence of transient objects, each with ever-changing context. These objects can be physical and/or virtual entities, and may enter or leave the system freely. For example, a commercial building application might want to monitor occupancy data and energy usage in its public lobby – it would want to achieve this without requiring every person to register

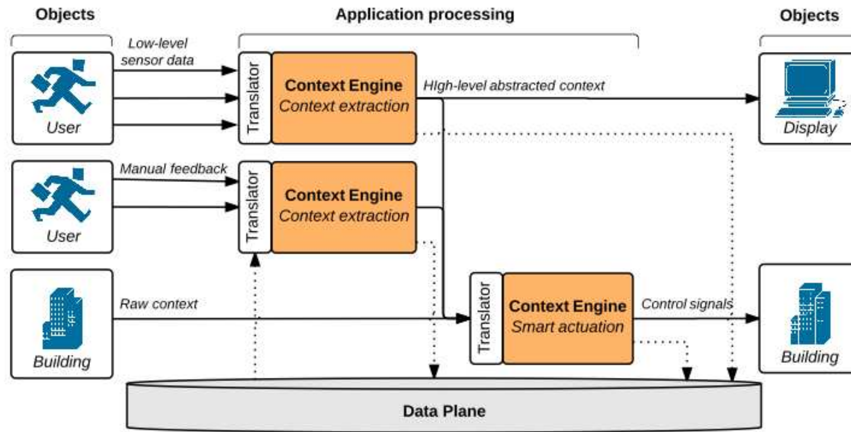

Fig 1. System architecture overview showing major components in bold, with implementation-specific examples italicized

explicitly when they appear, but instead collect data as it becomes available to the context engine. A complementary context-aware application on the users' side would provide location data relative to the building (either in the building or out). We use an ontology-driven data representation that allows sources to provide data at-will in a format that is understandable to context-aware applications. In the occupancy example, the building-relevant location data is only generated from the user's app when it detects the proximity to the building (via GPS, beacons, etc.) and only consumed by the building's app for the users who are generating this data. We also specify the interface for context translation, which allows the applications to understand and modify their scope of responsibility as their environment changes, and drive their output.

The context engine is implemented as a base model. The major functionalities - enforcing the ontology, determining that an application can process the input data, and providing output/actuation – are implemented, and individual applications inherit it. They are only required to provide the application-specific reasoning that drives the output. The ontology and the context engine base must be designed hand in hand, as the design decisions made in one affect the other. With that in mind, we first outline the system architecture, then describe the ontology-driven data translation layer, and dive into more detail on the most important system component - the context engine.

A. System Architecture

Fig 1 depicts the overall design of our system: one or more context engine instances (shown in orange) that ultimately transform input data into output data to be consumed by actuators or other applications. In the process, the system translates relatively raw sensor data into incremental stages of higher-level *context variables* associated with the objects (the physical and virtual entities) in the system. The exact use of this output data is application-dependent - as the figure suggests, an object could simply display contextually relevant data, or a building could take the processed context data as control signals to perform actuation.

As a high-level overview, each context engine instance has a scope of objects it takes input from, and a (possibly overlapping) scope of objects it affects with its output. Each of these objects has context variables associated with them. These variables make up the lowest-granularity pieces of data in the system, and are ultimately the inputs and outputs on which the engines operate. An ontology-driven context translator performs the formal mapping of data into the scope of objects and their constituent context variables, defined in the next section.

The actual transformation of the input context to the output context is application dependent. For example, an activity detection context application may consist of a preprocessing context engine instance that associates a person's (the object's) changed GPS location (the constituent context variables) and outputs an update to that person's abstracted location (e.g. home, work). A second context engine instance will watch for changes to a person's abstracted location and use machine learning to probabilistically reason on his/her activity, updating the perceived activity (and possibly a confidence threshold) associated with that person. Any and all of these intermediate and final context variables, all associated to the same person object, can be in turn used by other context-aware applications (e.g. residential building automation).

As in the above example, the output of each context engine instance is context – specifically, one or more context variables associated with certain objects. While Fig 1 shows a tree-like set of dependent context engines, the system of context engines can be connected as a graph, with the output of one or more context engine instances composing a subset of the input scope to another context engine. Alternatively, it can be the context change that directly drives actuation, or the interpreted and abstracted context that can be used by other applications (the building automation in the above example). Output can also be used to update the input object itself, representing feedback, and driving a refinement of the object's known state. In the activity example, the person's abstracted location and his/her activity represent intermediate and final context engine feedback to the original state.

The next sections expand on the mechanisms that compose the system: the ontology-driven context translation and the context engine structure.

B. Ontology-driven Context Translation

The ontology choice is important to the context engine, as it provides the structure of communication between the external sensors and the context-based reasoning within. Moreover, it connects the new middleware to the existing infrastructure, so it should both accommodate current pervasive sensing implementations and future technology. Additionally, as a dynamic environment, the IoT requires both the context engine and an ontology-driven interface to adapt to missing and newfound data.

Objects in the system may be physical or virtual entities that are associated with at least one piece of context data. As mentioned in Section II and [1], the Object-Role Model associates context to objects, which describe their relationships to the environment in different ways. We describe these pieces of information by *context variables*.

Context Space (S): The context space S of any object in the system is a set of context variables $\{n_1, n_2,\dots n_m\}$, where n is an element from all the possible context variables. In essence, S represents all the variables that can be applicable to an application. This meets the requirement of identifying the context of an object and represents a persistent set of applicable variables.

Context Dimension (D): the context dimension of an application, *D*, is a subset of *S* that represents the *minimum* set of context variables that the application can use to process an iteration. As the sensor sources (and subsequently, the context variables) available in an environment change, different context dimensions can become relevant or irrelevant. For example, more variables could lead to better learning, while fewer variables could be a less accurate but sufficient set for reasoning.

As illustrated in Fig 2, various sources of context data are fed into the context engine, first passing through the data translation layer at its interface. This ensures that all context data from both low-level devices and pre-translated data from a wider data plane comply with the unified data representation. They form the *context space* available to the application running on this context engine. The application specifies a list of context variables it needs to do processing, filtering its *context dimension* out of the available *context space*.

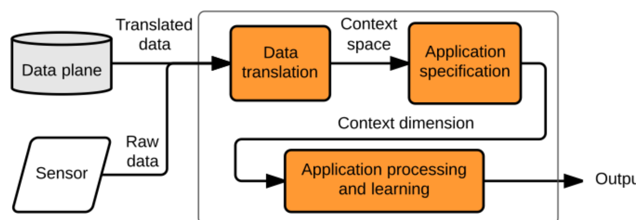

Fig 2. Flow and transformation of low-level data to high-level context output for a single context engine

A powerful feature of this data-driven context engine is the ability to *compose* various context variables associated with the same object. For example, a healthcare application may reason on collected posture information on its users, so it lists user posture data in its *context dimension*. That is, it will ignore a user in the system who only reports GPS data to applications. However, if such a user gains a wearable sensor and chooses to publish it through their personal context dimension, this new posture sensing data will appear in the *context space*. As the healthcare application scans for available data, this new user will appear in its scope, and it can start reading and reasoning on their posture information.

Naturally, this raises concerns with data accessibility, security and privacy. The context dimension is simply the way that entities in our current system (objects or applications) specify the data they are willing to share. Related works in the field of operating systems for distributed and ubiquitous computing have proposed more mechanisms to handle each of these issues that may be implemented alongside our data flow. In fact, one of our case studies (Section V.B) was implemented on a research data storage infrastructure [18] that would provide privacy and security independent of our system.

By translating and operating on data according to the translation model, we can fulfill the required properties of context-aware applications listed in Section II:

1. **Adaptability:** Context engines can operate on changing sets of data by updating their context dimensions with new variables in the context space.
2. **Distributed Data Aggregation:** The data translation interface to the context engine ensures compliance with the ontology, written in a web-standardized format. A current example would be today's standard format for web-based documents - XML.

1. **Data Flexibility:** A combination of the hierarchical object-role model and a strong data format allows applications to accept and exchange data whether it describes low-level sensor input or high-level context.

3. **Reasoning:** The engine enforces compliance with a unified data representation at the interface to the external system. Internal processing logic is still flexible and left up to the application developer.

C. Context Engine

The context engine is designed for the system in Section IIIA and encompassing the ontology-driven data processing outlined in Section III.B. An overview of the system is outlined in Fig. 1, to meet the following requirements of context-aware IoT applications:

1. **Context Identification:** each application must be able to determine the context variables relevant to itself.
2. **Flexibility of Reasoning:** when different sets of context data are available, the application must be

able to perform different types of processing.

3. **Extensibility:** as system and environment conditions change, the application should be able to be changed with minimal interruption of service.

Context identification reflects the property that is found in other middleware implementations – the ability to determine an application's needs – through explicit object identification or monitoring [5], [11]. We aim to maintain this property, but also add *flexibility of reasoning* and *extensibility*, neither of which are similarly represented in corresponding middleware. These two features enable more powerful improvements to applications through the context engine. For example, the ability to perform machine learning-based classification of context variables and automatically deriving context dimensions for applications reflects *flexibility of reasoning*. Similarly, online learning and modification of application behavior to incorporate new input and output context dimensions reflects extension.

Context Identification: Invoking the data translation layer's specification from the above section, each object and application has one or more input context dimensions associated with it. These are defined at application creation time, as it is tightly coupled to the application's processing, and specified as a list of objects and their relevant context data. The context engine continuously queries the objects for changes, and composes the constituent context variables required for the application. When the data collected matches any context dimension to completeness, the application can process and reason on it. While some subset of an application's context dimension must be specified at creation time, our context engine implementation allows this to be changed at runtime through the extensibility property below. Other middleware implementations treat this process as static – defined only at creation-time. However, as IoT applications are operating in a dynamic environment, it is only natural that their context can also be changed.

Flexibility of Reasoning: Flexibility of reasoning is defined by the data translation layer as a mapping of different input dimensions to the same output dimension. Practically speaking, it allows the application to respond to different types of input, making it adaptable. For example, a location-tracking application may use GPS coordinates by default, but defer to higher-accuracy indoor localization data when available (i.e. inside a building). By definition, the ontology-driven translation enables this, as the application can literally specify multiple input context dimensions, each of which specifies a different path to take to achieve the output. This enables a well-designed application to take advantage of new or additional context that may emerge about the objects in its input dimension.

Extensibility: We define extensibility as the ability to increase the input and output context dimensions with new objects and context variables. Unlike adaptability, this is an online, or post-creation, property. Practically speaking, this means that new objects and their constituent variables can be incorporated into the application. This also implies a fundamental change in the functionality of the application, since it must now incorporate and process based on entirely new data. One example is the building occupancy detection application that now has to comprehend and process a new means of a user reporting occupancy (e.g. presence on the local WiFi network). Our infrastructure provides multiple methods to achieve this goal:

1. *New context engine instances:* The simplest approach is to extend the code of the existing context engine, updating its dimension and functionality to handle the new data. This has the upside of simplicity, as the system simply replaces an existing engine with a new version. However, it requires an interruption of service while the original application is supplanted. Fig 3 below reflects this, as the new context engine uses a superset of the original input and output dimensions.

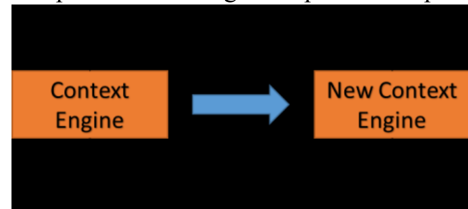

Fig 3. Replacing the original context engine with a new implementation to extend functionality. (A_1 U A_2) and (B_1 U B_2) reflect a superset of the original input and output dimensions, respectively.

2. *Parallel context engine instance:* A modification of (1) is a new context engine instance that that translates a different input context dimension into the redefined application's output dimension fulfills the extensibility requirement Fig 4(a). The benefit compared to 1) is a simpler secondary implementation, as it only deals with the new case while the original context engine handles the main functionality. It also maintains the processing and functionality of the original application uninterrupted. A subset of this case is an extension of the original processing in a completely independent function. This is highlighted in Fig 4(b), where the new context dimension exists in parallel to the original, and updates its output dimension independently.

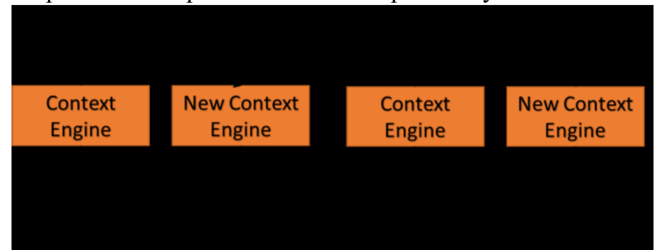

Fig 4. Online changes to an existing context engine instance, where (a) reflects dependent processing changes, and (b) reflects independent changes.

3. *Pre- & post-processing:* A subset of the extensibility definition is when only the input dimension or the

output dimension needs to change. While both the above solutions handle this (simply set A_2 or $B_2 = 0$). However, in the practical case where this change reflects pre- or post-processing (e.g. new context variables can be refined into a more accurate representation of the original context variables), we can simply add an upstream (preprocessing) and/or downstream (postprocessing) context engine instance. Fig 5 below reflects this case. The additional context engines demonstrate a refinement of the original input (a) and the original output (b) via feedback from the additional data.

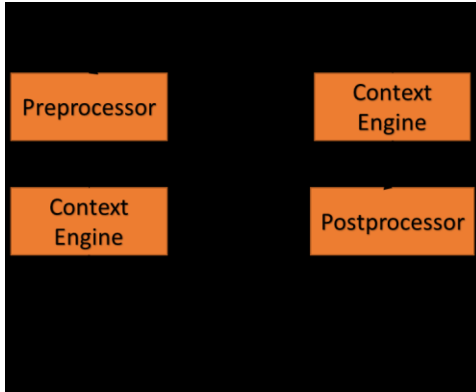

Fig 5. Online preprocessing (a) and postprocessing (b) of a context engine.

The extensibility cases above highlight an issue of online modification of functionality: data consistency and coherency, as multiple context engines can write to the same locations. While the issue itself falls out of scope of this paper, it is a well-understood problem in distributed computing. Several solutions, including locks and tokens can handle consistency independently of our middleware layer, and these are mechanisms that should be implemented in other parts of the IoT infrastructure [17].

The following section describes two distinct implementations of the context engine, highlighting both a machine learning application and an adaptable end-to-end system demonstrating sensing, preprocessing, and actuation with an IoT backend.

IV. System Implementation

The context engine consists of an ontology-driven data translation layer and a base context engine implementation. The former is a source-agnostic interface that ensures incoming data representation fits the ontology and parses it into an acceptable data format – in our case, XCML. The latter is the simplest complete context engine for a generic application. All application-specific context engine implementations inherit from and, if appropriate, extend this base implementation.

A. Data Representation

We employ an extended implementation of XCML as our ontology-driven data representation. XCML is a markup-based (XML) implementation of the Context Modeling Language. It has built-in support for context space and context dimensions. However, it assumes context data is always represented with a flat hierarchy of simple data structures. Meanwhile, context-related information often consists of nested data types or tuples - GPS coordinates, for example, should come in pairs of values for latitude and longitude respectively; the GPS data type itself could also conceivably be nested within a larger data structure, next to fields that contain data from other localization schemes. Thus, complex data structures are required to have a 1:1 translation of internal and external data representations. As recommended in [1] [6], we use a JSON model to support the internal representation, and we extended the XCML paradigm to provide the 1:1 translation:

1. *Lists:* Lists are identified in the context dimension by adding the attribute `type="list"`. This signals the parser to treat the encapsulated data structure as a repeated type within the parent, not as a top-level entity. In the context data, the corresponding tag will simply be followed by a list of the data in the appropriately specified type.
2. *Dictionaries:* similarly, dictionaries are specified in the dimension with the attribute `type="dict"`. Unlike lists, however, the encapsulated type information is treated as a set of key-value pairs.

The addition of lists and dictionaries strengthens the XCML language, and we further extend it to handle nested combinations of all the constituent types, and a unique translation to the internal representation and vice versa.

B. Data Translation

The translator enforces a data format required by the ontology, as well as the context required by each object and application. Our revised XCML format provides both the ontology's specification and the actual data format.

The application developer specifies XCML data in locations accessible to the context engine. This XCML data identifies objects and their supported context variables, and application context dimensions. Fig 6 (top) depicts an example of the web browser object and the associated context variables associated with it in the dimension. The translation of this dimension into the actual data is accomplished by scanning all objects that match the particular context dimension name (e.g. Browser), and capturing objects that contain all the dimensions required for this application. In the example, the context-state tag holds one browser that was found in the discovery process, with the version and agent data populated. Changes or updates to these objects will be pulled by all context engine instances that monitor these objects. The case studies in the next section illustrate more specific examples of this data is stored, retrieved and used. While this is a relatively simple example, an application's context dimensions will be stored in such an XML file, containing the myriad of context dimensions that determine a complete set of input data required by the application. Recursive parsing translates both an XCML file (external representation) into a

corresponding JSON structure (internal representation) and vice versa.

```
<xcml:context-dimension name="Browser">
  <xcml:context-key name="agent"/>
  <xcml:context-key name="version"/>
</xcml:context-dimension>
<xcml:context-state name="old_IE" dimension="Browser
  <xcml:context-property key="agent" value="IE"/>
  <xcml:context-property key="version" value="6"/>
</xcml:context-state>
```

```
{"old_IE":
         {"agent": "IE"
          "version: "6"
         }
 }
```

Fig 6. Sample XCML (external) and JSON (internal) parsed context

Fig 6 above shows the same data translated into the internal and external representations. The top of the figure shows the XCML representation: the context dimension outlines the requirements for each complete piece of context. From an initial file that had much more raw data, the parser was able to isolate and filter one unique piece of context (`xcml:context-data`). The bottom of the figure represents the internal representation: a direct JSON translation of the context data.

C. Context Engine Base

The different context engine instances that make up an application (see Fig 1) serve different purposes: data translation and context filtering, the actual application reasoning and processing, and actuation. However, they all share and extend the same base context engine implementation.

Every application has associated XCML data representing its context dimension(s). The translator from the previous section scans the file into the JSON internal representation, and the context engine searches its known data space for objects that match each dimension element. The data space can be any representation accessible to the application: in the case studies in Section V, we read data from both a file and a key-value store implemented on an IoT data plane [18]. The actual processing is application dependent, and is left as an abstract function (*runApplication*) – a sandbox for application-specific reasoning. When an application inherits the base implementation, it is required to populate its processing logic in this function. As long as the application's context dimensions are defined, the base framework will scan the specified data space for objects that are specified by the dimension. When any subset of the collected data matches a context dimension completely, the *runApplication* method is triggered and the application can perform its processing.

The context engine base and the data translation layer are implemented in Python. We leverage the XML.ElementTree libraries to read and write the XCML representation.

In the next section, we investigate two such applications, which take data from different sources and locations (from file and from an IoT backend), perform different types of reasoning, and produce different actuation.

V. Case Studies

In this section, we describe two different examples that illustrate our context engine concept – a step counting application that uses machine learning over a Bayesian network, and a lamp actuation application that is implemented on an end-to-end IoT infrastructure and demonstrates seamless application extensibility and ease of implementation.

A. Step counter:

Fitness trackers currently dominate the wearables market [19], with many interface options available to the user, including smartphone and desktop applications, and development APIs opened to development enthusiasts. In fact, users often wear devices that report partially redundant information, even if they were designed to collect disparate data at a low level. This example collects user activity from two independent devices - a Fitbit Flex step counter [20] and an Android smartphone running the mobile application Moves [21]. While both data streams report a user's step count, they arrive at that conclusion in different ways and with different reliabilities. We have a simple goal of obtaining an accurate daily step count for a single user based on these two data streams, by learning when and where to trust one data stream over the other.

1) Input Data

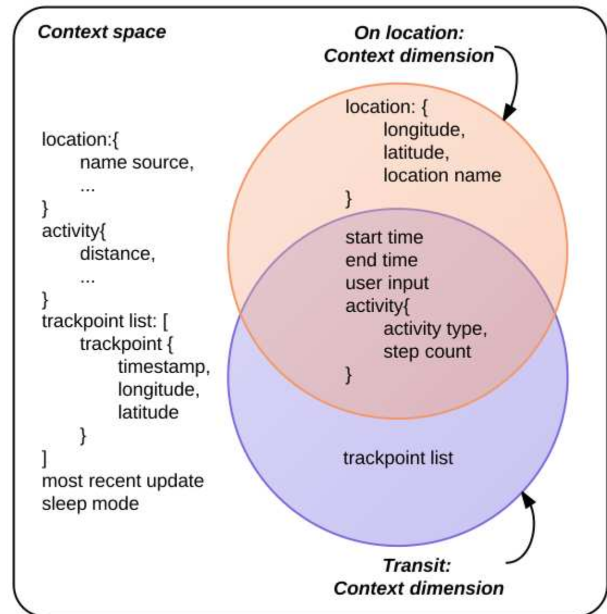

Fig 7. Relevant variables filtered from the context space into an application's context dimensions

Fitbit reports minute-by-minute step counts for a user, based on accelerometer readings. The data traces are simple. Each piece of context data only contains a start and end time for the interval (currently fixed at 1 minute) and the number of steps counted. While the device actually reports more, such as a user's sleep mode and inferred

activity levels, our application does not need those pieces of data and thus leaves it out of the context dimension.

```
<xcml:context-dimension name="Fitbit">
  <xcml:context-key name="startTime"/>
  <xcml:context-key name="endTime"/>
  <xcml:context-key name="stepCount"/>
</xcml:context-dimension>
```

Fig 8. Context dimension for simple input from Fitbit

Moves traces include higher-level activity readings such as semantic location names, type of motion, and a step count if the user is walking. These readings are inferred from a variety of low-level sensors and crowd-sourced data (smartphone accelerometer, GPS, social location check-in service, etc). The traces are divided into "segments", where the user is either sedentary, moving from one point to another, or moving around within one location. Each segment is bookended by a start and end time, but it may also exercise different combinations of context keys, some of which are organized into nested structures, as shown in Table 1. When the user is at a location for an extended period of time, it records the GPS coordinates, and may include a semantic locality name depending on availability. The semantic location name may be estimated from Foursquare (a web-based social location service) [21], or manually entered by the user - this naming source is also recorded and can imply a measure of confidence in the location. If the user is in transit, Moves records instantaneous positions in a list of timestamped GPS coordinates, called "trackpoints".

```
▼<xcml:context-key name="trackPointList" type="list"
  ▼<xcml:context-key name="trackPoint" type="dict">
      <xcml:context-key name="timestamp"/>
      <xcml:context-key name="longitude"/>
      <xcml:context-key name="latitude"/>
    </xcml:context-key>
  </xcml:context-key>
```

Fig 9. Truncated context dimension for input from Moves, showing a list (trackPointList) of complex keys (trackPoint) containing simple keys (timestamp, etc)

The user manually collects the verification data for an actual step count. The start and end timestamps are required to align collected data with Fitbit and Moves data, and the unique context variable here is "user input". When the boolean "user input" is *True*, it implies highest confidence in the user-supplied data.

```
<xcml:context-dimension name="ManualStepCount">
  <xcml:context-key name="startTime"/>
  <xcml:context-key name="endTime"/>
  <xcml:context-key name="userInput"/>
  <xcml:context-key name="comment"/>
  <xcml:context-key name="steps"/>
</xcml:context-dimension>
```

Fig 10. Context dimension for manual step counts collected directly from a user

2) Context Engine Implementation

From observation, we have a sense of systematic errors from Moves - for example, the GPS tracking has high latency when waking from sleep and misinterprets movement upon "catching up" to tracking. Intermittently, it derives the step count from the distance traveled divided by a universal average stride length (thus undercounting steps for a shorter person). Both devices are susceptible to losing battery, losing data due to connection or cloud service failures, or simply being misplaced by an absent-minded user. Fitbit has a tendency to misinterpret miscellaneous movement (typing, doing dishes, etc.) as steps, while missing less easily discernible steps (carrying groceries, hands in pocket). By taking both devices into the context-aware application, and leveraging the fact that Fitbit and Moves give slightly overlapping but different information, we have the opportunity to weigh their data given contextual reliability and fall back to one if the other goes offline.

We classify segments of a user's day coarsely based on data from Moves. When the user is in one place, the phone records data as a "location" and associates a list of activities with it. However, when the user is in transit, the system records a single movement "activity" (transit, walking, cycling, etc.) with a series of "trackpoints" tracking the movement. The two *context dimensions* in Table 1, both extracted from the same *context space*, reflects this. The two context dimensions allow the application to parse and reason across two different scopes of data.

The Bayesian network in Fig 11 describes the relationships between observable data from Fitbit and Moves, each node representing some variable summarizing the user's activity. Using the specification from Fitbit's API, we classify the step counts per minute into low, medium or high activity levels. The context engine trains the network on each incoming data segment and constructs the associated conditional probability table based on whether Fitbit or Moves is more accurate for each segment (relative to the ground truth). The dependencies of the nodes are determined by the classification of activity levels (low, medium, high), and the perceived Moves activity (on location, walking, or using transportation). Each instance where Fitbit (or Moves) agrees with the ground data (within margins of the activity level thresholds) increases the weight of the edge leading from that node to the "Fitbit is accurate" node (or "Moves is accurate" node).

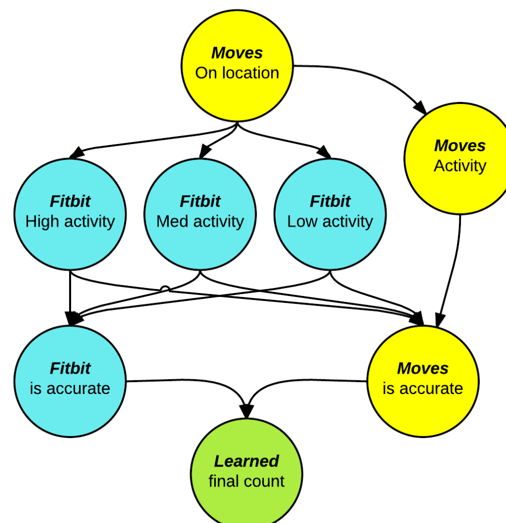

Fig 11. Bayesian Network to learn accurate step count from Fitbit and Moves data.

In Fig 11, the edges connecting dependencies were manually assigned, and only their weights were learned. Without *a priori* knowledge of the relationship between Fitbit and Moves, the Bayesian network would start as a fully connected graph, and the learning algorithm would eventually prune the edges which do not in fact connect dependent nodes.

3) StepCountEngine results

We compare the accuracy of these three data streams - Fitbit, Moves, and estimation learned on Fitbit and Moves. The edge weights of the Bayesian network are trained on two days' worth of data. Since each day is naturally divided into a different total number of segments depending on how often the user changes activities or locations, this ranges between 37-45 segments. After the learning phase, the cross product of all these variables gives a confidence for each data stream that selects the most trusted source for each segment. The final daily total of step counts is compared to the "ground truth" for that day's total, and the "accuracy" represents how closely the estimated step count falls relative to the actual number of steps taken. We define accuracy as the probability of designating the output as "correct", under a Gaussian distribution. The mean is selected as the "ground truth" data and the standard deviation set such that the +/-15% range has a 90% accuracy. This margin is given because small discrepancies in the absolute number of steps counted across a day (which ranges in the thousands on average) should be reasonably expected.

Fig 12 shows the performance of our learning algorithm on a small training set, for three representative days. In Sample 1, even though both Fitbit and Moves are grossly inaccurate in counting total steps for the day, the other contextual data they provide about a user's location and activity type can help greatly in learning which one to trust for a particular segment in the day. Thus, for each segment, as long as one of them reports accurate data, and the learning algorithms picks the correct one, the final count can still be very accurate.

In Sample 2, the Fitbit stream is highly accurate, while Moves is not. While the node weights of the Bayesian network represent confidence in a data stream, they do not guarantee accuracy – by correctly choosing to trust Fitbit in most segments and Moves in a minority of segments, the final step count is still highly accurate. On the other hand, the penalty of trusting in the wrong stream can negatively impact the learned result, as shown in Sample 3. Such an "anomalous" day may include long periods of time spent underground where GPS localization is ineffective, or where the user is engaged in vigorous and repetitive activity while standing relatively still, such as organizing equipment in a lab environment.

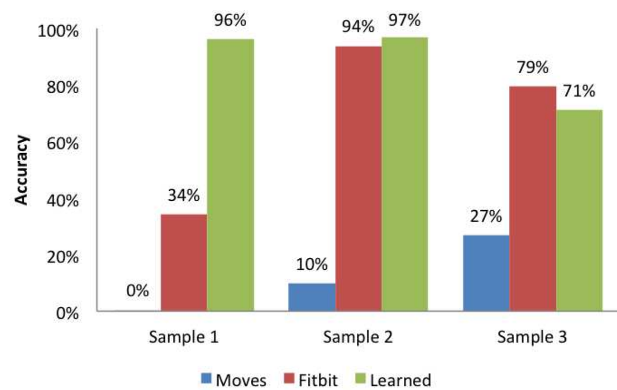

Fig 12. Accuracy of Fitbit, Moves, and Learned step count for 3 days

By learning when to trust and when to discard particular data streams based on the context of that data collection, we can produce an output stream with accuracy much higher than either of the individual sensor streams alone. In this case study, we see an average 60x accuracy improvement over using a single stream of data when learning over just 2 days.

B. Automated Lamp Actuation:

Smart buildings are a quintessential Internet of Things application, and simple hard-wired, occupancy-based lighting automation is already ubiquitous. However, developing technology like the web-connected Philips Hue™ lamp [22] exemplify the IoT aspect of smart buildings: an actuator (color-changing lamp) whose state can be read and modified through the Philips API. While users can access the interface manually, implementing a context-aware application based on their color preferences and proximity to the lamp is the IoT implementation of the application. For this implementation, we design a context-aware application that will change the lamp to a user's preferred color when they are sitting in the room where the lamp is installed. When multiple people are sitting, the lamp reaches an intermediate color between all the occupants' preferences.

1) Input Data

This application requires data about whether or not a user is seated in the room, and their color preferences. The information about a user's orientation (seated/standing) is determined by a wearable activity sensor. The XCML context dimension for the user is shown in Fig 13 below. A preprocessing context engine instance filters all the applicable data for a person, returning only the relevant information: the id of the person, whether or not he/she is seated, when the person changed state, and his/her color preference.

```
<xcml:context-dimension name="Person">
        <xcml:context-key name="id"/>
        <xcml:context-key name="timestamp"/>
        <xcml:context-key name="isSeated"/>
        <xcml:context-key name="color"/>
</xcml:context-dimension>
```

Fig 13. Context dimension for each person in the application.

Similarly, the application requires the status of each chair in the room: the id of the chair, whether or not it is occupied, and the time at which the chair was occupied. The relevant XCML context dimension for each chair in the room is shown in Fig 14.

```
<xcml:context-dimension name="Seat">
  <xcml:context-key name="id"/>
  <xcml:context-key name="timestamp"/>
  <xcml:context-key name="isOccupied"/>
</xcml:context-dimension>
```

Fig 14. Context dimension for each seat in the application.

2) Context Engine Implementation

The context engine implementation, SeatApp, operates on the people and the seats in the room at any given time. Thus, the context dimension for the application itself is as shown in Fig 15 below:

```
<xcml:context-dimension name="SeatApp">
        <xcml:context-key name="Seats" type="list">
                <xcml:context-key name="Seat" type="dict">
                        <xcml:context-key name="id"/>
                </xcml:context-key>
        </xcml:context-key>
        <xcml:context-key name="Persons" type="list">
                <xcml:context-key name="Person" type="dict
                        <xcml:context-key name="id"/>
                </xcml:context-key>
        </xcml:context-key>
</xcml:context-dimension>
```

Fig 15. Context dimension for the automated lamp (SeatApp) application

The specification identifies that the application operates on a list of seats and people that are deemed to be currently in the room at a given time. The id for each seat/person is the key on the data store to look up for each source. The critical part of the application is that the list of seats and people can change at any time.

3) IoT and Hardware Infrastructure

The wearable orientation sensors and the pressure sensors for the chairs are developed and deployed by UT Dallas [23], and the output is preprocessed into the XCML-appropriate format using impulse matching of positive signals [24].

As a true end-to-end context-aware IoT application, the SeatApp was implemented on an IoT infrastructure. The Global Data Plane (GDP) provides a flexible web-based data store, implementing key-value data access through a HTTPS REST protocol [18]. The GDP readily accepts the XCML context information from the previous sections, allowing the application to read, write, and modify the system information.

4) Application Processing

The application runs on a continuous loop. At each iteration it is driven by the seats and people found on the data associated with the above dimension. At any time, a person or seat can be manually entered into or removed from the known system by making corresponding changes to the context dimension. These changes will be seamlessly incorporated into the next iteration.

The application logic maps seated users to occupied chairs by matching their timestamps. The color preferences for each seated user are averaged together using the Hue color palette [25] to provide an amenable median color for all present users.

The output specification for the lamp simply identifies a lamp id, the color to set the lamp to, and the timestamp recording the change (Fig 16). While this output is generated by the context engine and consumed by the lamp, it is easily conceivable to use this information as context for determining building occupancy by other applications: HVAC and security logic can make use of the knowledge of which buildings are occupied and by whom for further automation of the building. Because the output conforms to the XCML ontology, the dimension information and data can be directly used by other context engine implementations.

```
<xcml:context-dimension name="Hue">
  <xcml:context-key name="id"/>
  <xcml:context-key name="timestamp"/>
  <xcml:context-key name="color"/>
</xcml:context-dimension>
```

Fig 16. Hue Lamp output specification in an XCML context dimension

5) Uninterrupted Application Extension: GPS data

While the automated lamp example is a good example of the context engine in use for an IoT application, other ontologies and middleware can accomplish the same end result. What separates the context engine is the ability to seamlessly extend the functionality for new data types and/or sources without unnecessary re-implementation of the application or even interruption of service. With that in mind, we extend the above example: if we add GPS information to the seats and people, we can remove the manual updating of the list of seats and users in the room. However, since the functionality of the application remains the same, we can simply implement a new context engine to update the SeatApp data that it consumes (Fig 15).

```
<xcml:context-dimension name="GPSLoc">
  <xcml:context-key name="id"/>
  <xcml:context-key name="lat"/>
  <xcml:context-key name="long"/>
</xcml:context-dimension>
```

Fig 17. GPS location context dimension, keyed to the id of the object.

The new context engine implementation, SeatAppUpdate, serves as one of the “upstream” engines on the left of Fig 1. It verifies a list of all users and seats that have location tracking, and verifies if their GPS location is in the bounding box of the room. Using the same object IDs from Fig 13 and Fig 14, SeatAppUpdate searches the data associated with that user for GPS information stored in the XCML data mapped to his/her ID (Fig 17). It performs a simple bounding of the user’s location compared to the room’s location, and modifies the SeatApp XCML data to add/remove the user as appropriate. By the simple addition of a single upstream context engine instance, the application now automates user registration. Note that users can still manually register

themselves or seats as in the original program, but the automation runs in parallel, updating the input data as appropriate. As soon as SeatAppUpdate is executed, SeatApp picks up and handles the automation at its next iteration without needing to be interrupted.

Note an additional Fminor but extremely crucial detail: the ontology defines the format of the GPS location (Fig 17), but does not specify the source of the information. A user's GPS is thus allowed to come from any appropriate source (e.g. smartphone, wearable GPS, beacons, etc.) as long as it can be translatable to latitude and longitude. This gives the new implementation enormous flexibility. A user (and a chair) can allow his location information to be available to the system through any means available. In the practical implementation, the user's GPS location is retrieved from the smartphone, whereas a chair's GPS is derived whenever it was placed in close proximity to a beacon.

6) Results

As a full end-to-end implementation of a user-driven smart lighting application, SeatApp represents more than just a working example: it covers all the requirements of the context engine as specified in Section 3, which none of the related work managed to accomplish. It performs *context identification* in the presence of heterogeneous sources pushing data to independent backends. It provides *adaptability* by implementing different context dimensions for manual and automated detection of locality. Finally, and unlike any of the previous works, it provides *extensibility* in a seamless manner: extension of the original application to new context (GPS) was accomplished without interrupting the application itself. This is crucial to the practical reality of the IoT, as it demonstrates the ability to incorporate new and replacement technology in an incremental and low-overhead manner.

An additional metric for gauging low-overhead is the lines of code required to implement an application. With the exception of the base context engine code and the implementation of the logic, the engine-specific application code was only 119 lines, or under 29% of the total context implementation. The infrastructural addition of the GPS was a mere 30 lines of code, bringing the total context overhead to just over 20%. That is, once the application's base implementation is established, incremental changes to the infrastructural overhead are minimal even as the complexity of the application logic grows, significantly reducing the ultimate overhead of the context engine.

VI. Conclusion

The Internet of Things represents the next iteration of ubiquitous computing, incorporating heterogeneous sensing and actuation in a web-based backend. However, the onset of significant commercial development of IoT devices communicating with divergent backends and in constantly-changing formats severely complicates the main goal of the IoT: context-aware computing. Upon reviewing the related

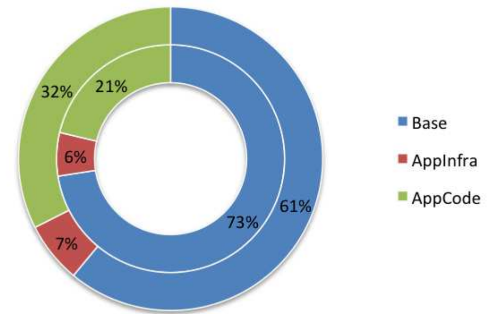

Fig 18. Breakdown of codebase by lines of code. Minimal changes are required to the application-specific infrastructure code (AppInfra) to handle the addition of GPS data. Arbitrary application code (AppCode) expands while the base implementation (Base) remains the same

IoT work, we found a key component of the new IoT middleware missing: the ability to unify and operate on ever-changing sources and context in a low-overhead manner. To resolve this issue, we developed the ontology-driven context engine. Leveraging and expanding the XCML context ontology, we unified data translation, filtering, and preprocessing into a format readily readable and expandable by context engine implementations. We developed a methodology to implement context-aware applications: upstream context engine instances for flexible context preprocessing and extension, and downstream instances for application logic and actuation. Using a base context engine, we implemented two different applications: one that demonstrates the ability to learn and follow different paths of reasoning based on available data, the StepCountEngine demonstrates up to 3x improvement in context accuracy. The second application is an end-to-end context aware implementation on IoT infrastructure using heterogeneous sources, and demonstrates the ability to extend the application with new sources without interrupting the original application, all the while attributing only 20% of the total application code to incorporating the new infrastructure.

References

[1] C. Perera, A. Zaslavsky, P. Christen and D. Georgakopoulos, "Context Aware Computing for the Internet of Things: A Survey," *IEEE Communications, Surveys, & Tutorials,* pp. 414-454, 2013.

[2] M. Friedewald and O. Raabe, "Ubiquitous computing: an overview of technology impacts," *Telematics and Informatics,* vol. 28, pp. 55-65, 2011.

[3] J. Hammer and T. Yan, "Poster: A virtual Sensing Framework for Mobile Phones," in *Proceedings of MobiSys*, 2014.

[4] J.-H. Hong, S.-I. Yang and S.-B. Cho, "Conamsn: A context-aware messenger using dynamic bayesian networks with wearable sensors," *Expert Systems with Applications,* vol. 37, no. 6, p. 4680–4686, 2010.

[5] S. K. Madhu, V. C. Raj and R. M. Suresh, "An Ontology-based Framework for Context-Aware Adaptive E-Learning System," in *International Conference on Computer Communication and Informatics (ICCI)*, 2013.

[6] S. Staab and R. Studer, Handbook of Ontologies, Springer Science and Busines, 2010.

[7] Lee and K. e. al., "AMC: Verifying User Interface Properties for

Vehicular Applications," in *Proceedings of MobiSys*, 2013.

[8] P. Rashidi, D. Cook, L. Holder and M. Schmitter-Edgecombe, "Discovering activities to recognize and track in a smart environment," *IEEE Transactions on Knowledge and Data Engineering,* vol. 23, no. 4, pp. 527-539, 2011.

[9] M. Rudary, S. Singh and M. E. Pollack, "Adaptive cognitive orthotics: combining reinforcement learning and constraint-based temporal reasoning," in *Proceedings of the 21st International conference on Machine Learning*, 2004.

[10] H. Chen, T. Finin and A. Joshi, "An Ontology for Context-Aware Pervasive Environments," *The Knowledge Engineering Review,* vol. 18, no. 3, pp. 197-207, 2004.

[11] T. Gu, X. Wang, H. Pung and D. Zhang, "An Ontology-based Context Model in Intelligent Environments," in *Proceedings of communication networks and distributed systems modeling and simulation conference*, 2004.

[12] W. e. a. Wang, "A comprehensive ontology for knowledge representation in the internet of things," in *11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[13] M. Nebeling, M. Grossniklaus, S. Leone and M. Norrie, "XCML: providing context-aware language extensions for the specification of multi-device web applications," *World Wide Web (WWW),* vol. 15, no. 4, pp. 447-481, 2012.

[14] Google, "Google Now," Google, 2014. [Online]. Available: http://www.google.com/landing/now/. [Accessed 10 November 2014].

[15] S. Lee and K. C. Lee, "Context-prediction performance by a dynamic bayesian network: Emphasis on location prediction in ubiquitous decision support environment," *Expert Systems with Applications,* vol. 39, no. 5, p. 4908–4914, 2012.

[16] C. Perera, A. Zaslavsky, M. Compton, P. Christen and D. Georgakopoulos, "Context Aware Sensor Configuration Model for Internet of Things," in *12th International Semantic Web Conference*, Sydney, 2013.

[17] S. Bandyopadhyay, M. Sengupta, S. Maiti and D. Subhajit, "A Survey of Middleware for Internet of Things," *Recent Trends in Wireless and Mobile Networks,* vol. 162, pp. 288-296, 2011.

[18] J. Kubiatowicz and E. Allman, "Global Data Plane | SWARM," UC Berkeley, 2014. [Online]. Available: https://swarmlab.eecs.berkeley.edu/projects/4814/global-data-plane. [Accessed 30 October 2014].

[19] Juniper Research, "Smart Health & Fitness Wearables: Device Strategies, Trends & Forecasts 2014-2019," Juniper Research, 2014.

[20] Fitbit, "Fitbit Development," Fitbit, 2014. [Online]. Available: http://dev.fitbit.com/. [Accessed 30 10 2014].

[21] ProtoGeo Oy, "Moves For Developers," ProtoGeo Oy, 2014. [Online]. Available: https://dev.moves-app.com/. [Accessed 30 10 2014].

[22] Philips Inc., "Meet hue," Philips Inc., 2014. [Online]. Available: http://www2.meethue.com. [Accessed 20 October 2014].

[23] Y. Prathivadi, J. Wu, T. Bennett and R. Jafari, "Robust Activity Recognition using Wearable IMU Sensors," *IEEE Sensors,* 2014.

[24] N. Kale, J. Lee, R. Lotfian and R. Jafari, "Impact of Sensor Misplacement on Dynamic Time Warping Based Human Activity Recognition Using Wearable Computers," in *Proceedings of ACM International Conference on Wireless Health*, San Diego, 2012.

[25] Philips Inc., "Core Concepts: Philips Hue API," Philips Inc., 2014. [Online]. Available: http://www.developers.meethue.com/documentation/core-concepts. [Accessed 30 October 2014].

[26] S. Pantsar-Syvaniemi, K. Simula and E. Ovaska, "Context-awareness in smart spaces," in *IEEE Symposium on Computers and Communications*, 2010.