Seddacco: An Extensible Language in Support of Mass Collection of Health Behavior Data

Dylan Knowles
Dept. of Computer Science
University of Saskatchewan

Kevin G. Stanley
Dept. of Computer Science
University of Saskatchewan

Nathaniel D. Osgood
Dept. of Computer Science
University of Saskatchewan

{dylan.knowles, kevin.stanley, nathaniel.osgood}@usask.ca

ABSTRACT

Understanding human health behavior is fundamentally important to modern epidemiology. Increasingly, data on human health behaviors is acquired from telemetry systems initiated either by the user as part of the quantified self movement or by health researchers and public health agencies seeking to understand the behavioral determinants of health. Such measurement tools have traditionally been deployed on a hard-wired, case-by-case basis, but are beginning to provide broader configuration options to enhance reusability. In this paper we provide an analysis of the language requirements for configuring such a system and present Seddacco, an extensible application language that we have developed to meet those requirements. Seddacco applies metalinguistic abstraction and domain-specific language principles to improve configuration modularity, extensibility, flexibility, conciseness, and accessibility across the breadth of interdisciplinary health teams.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – *Specialized application languages*

General Terms

Languages

Keywords

Macroprogramming Language, Declarative Language, Domain Specific Language, Metalinguistic Abstraction, Behavior Measurement, Health.

1. INTRODUCTION

The development of cheap, readily available mobile wireless hardware has allowed researchers to monitor behaviors, activities, and interactions within areas, environments, and human populations. Although traditional methods such as surveys and interviews have generated significant insight, they are marked by a number of limitations, including observability, cost, and logistical challenges [3]. In recent years, these challenges have stimulated the rise of systems to provide rich, deep, and large sensed datasets on human behavior. New collection tools circumvent traditional shortcomings by acting as silent, less biased, ever-watchful, and less burdensome observers.

When collecting behavioral data automatically, custom hardware, smart devices, cellphones, and smartphones have been used [3]. These devices must be configured to meet each study's particular needs. This task is non-trivial and has spawned a variety of domain-specific programming languages and automatic configuration tools. The bulk of these languages, known as *macroprogramming languages*, are suited for monitoring large areas. For instance, TAG [11], MacroLab [5], and Semantic Streams [6] are languages suited for efficiently drawing information from a distributed network of collection devices. Other languages are suited to configuring individual devices that a participant would carry. These include the language that powers Healthopia [7] and the XML-like files of EpiCollect [8].

Any language or tool for configuring individual-centric mass data collection systems should aim to resolve, mitigate, or hide such systems' peculiarities from the study, yet expose enough features to enable a range of experiments. The system must be energy efficient, support a variety of collection modes, and have communication support. Flexibility and extensibility are key if additional data streams are expected or if protocols may be modified mid-study. The system should facilitate interaction with external devices, and should capture and respond to events of interest. Given the growing number of systems that support cross-linked collection from both sensors and respondent self-reporting, this small list is illustrative of the fundamental problem: there is enormous diversity in the features a configuration language must incorporate. Without specifying the features to be captured, designers may build tools that are unnecessarily restricted.

We explore and attempt to address this problem through two contributions. The first is a description of the requirements a configuration language must meet to be of general use in health applications. The second is a language designed to meet these requirements. We demonstrate how the language meets the domain requirements and how it meets the needs of the collection system on which it is based. We compare the language to others in similar spaces, and how the language might be integrated with existing technologies to form more general and flexible tools.

2. RELATED LITERATURE

Early developers of individual-centric mass data collection systems used low-level languages such as nesC [9] and TinyScript [10]. These languages experience limitations when attempting to build programs that are easy to use, understand, and modify and were followed by higher-level languages. TAG and AQL [11] and TinyDB [12] use an SQL-like syntax to acquire and report sensor readings. Abstract regions [13] and Hood [14] simplify network communication and data manipulation. Kairos [15] allows a programmer to define the global behavior of a sensor network through simple programming primitives.

Several languages were designed to facilitate interactions with domain-specific data abstractions. EnviroSuite [16] allows programmers to interact with abstractions tied to user-defined signals. Semantic Streams [6] allows users to query data collection networks for interpretations of data. EnviroTrack [17] allows users to track objects in an environment.

ATaG [34] converts specifications given in a high-level query language into smaller disseminatable programs. MacroLab [5] offers a vector programming abstraction similar to Matlab. Regiment [19] supports programs that rely on spatial and temporal factors via a signal abstraction.

Some tools exist to facilitate the creation of user-centric apps. Context Monitoring Queries (CMQs) [20] help applications identify context changes in collected data. CMQs inspired the syntax of Healthopia [7], which augments the language to resolve resource conflicts. EpiCollect [8] is a smartphone app that allows individuals to record data entries in the field. The user exploits an online form builder to specify data fields of interest, which generates on-device interfaces mapped to on-server databases. Hashemian [2] and Knowles [3] give brief descriptions of iEpi's configuration language in various stages of its development.

3. REQUIREMENTS ANALYSIS

Three primary components motivate the design of a macroprogramming language: the underlying fundamental use case, the capabilities of the supporting hardware and software, and general guidelines from the domain. The use case provides a set of features that are needed to meet the functional requirements of the task. The domain guidelines inform the final language structure and help determine the balance between efficient requirement fulfillment and ease of use. The hardware and software environment impose functional constraints.

3.1 Fundamental Use Case

Our fundamental use case relates to monitoring human spatial-temporal behavior, collecting information on when and with whom individuals were at specific locations, and what they were doing there. The task of monitoring behaviors is fundamentally a problem relating observations of health behaviors (e.g., physical activity, sedentary behavior, tobacco use, dietary intake) with the exposome [26][27] – the set of exposures to which an individual within a population is subjected. For example, long daily commutes may remove the time needed for physical activity, the nearest grocery store might be far away, or the respondent may be enmeshed in social networks exhibiting high prevalence of risk behaviors or environmental risk exposure. While elicitation of physical activity, dietary, location, and social proximity information is challenging and burdensome with traditional mechanisms, such information can be readily collected via sensor-based systems. Adding external data streams to a monitoring system, such as those generating heart rate and social network information, can provide information on the health of an individual or population. Participant responses given to contextually-triggered survey tools, can lend insight to sensor data by tagging temporally proximate context to the sensed behaviors.

Spatial monitoring requires both indoor- and outdoor-specific positional telemetry, as well as activity and social proximity sensing. The system must use these resources intelligently to avoid prematurely exhausting battery reserves. As data might also be drawn from connected off-device sensors or additional apps, the system must be flexible to incorporate external data streams. The system must also be able to handle data of a near continuous (e.g., location) and discrete (e.g., questionnaire responses) nature. The system should be able to conditionally trigger survey instruments (e.g., questionnaires), or enable certain sensor modalities. Given the sensitive and personal nature of the data collected, data should be processed on-device such that it is encrypted, obscured, or otherwise protected. Such a system should allow for reconfiguration (e.g., by issuing new questionnaires) throughout the study. Data must also eventually make its way to the client's (e.g., researcher, public health agency) servers.

3.2 Domain Requirements

The motivating example points to many items that an individual-centric mass data collection system should address. A broadly applicable configuration language in this space should therefore accommodate the following needs:

Data Management: Allow the system to collect, store, and upload data.

Internal Extensibility: Allow the system to integrate new features, e.g., new collection modes.

External Extensibility: Allow external applications and devices to be integrated into the system, where possible, should such applications and devices provide better data collection and processing facilities than the system itself.

Conditional Execution: Facilitate conditional execution based on the monitored participant's state to capture context-sensitive information that would be inefficient or infeasible to monitor.

Efficiency: Provide mechanisms for efficient battery usage, such that data collection is unlikely to cease before the end of the collection period.

Data Processing: Permit the post-processing of data to facilitate filtering and encryption.

Deployment Flexibility: Allow the system and its features to be configured for a variety of disjoint deployments, and to support reconfiguration.

Many of these features can be combined and generalized. Data management, internal and external feature exploitation, and data processing all represent information processing elements. Conditional execution and deployment flexibility require that the system be configurable. Conditional execution, when combined with efficiency, also requires that the system be configurable for temporal execution. These generalizations constitute a requirement set that the language should address:

System Extensibility: Allow a wide variety of information processing elements to be run. These include algorithms requiring sources both internal and external to the system.

Execution Specificity: Allow information processing elements to be augmented or restricted to precisely specify how they should execute in a given context.

Temporal Specificity: Allow information processing elements to be augmented or restricted to precisely specify when and in what contexts they should execute.

Several recurrent themes exist in the literature for implementing macroprogramming, domain-specific [21][22], and general programming language design:

Reusability: Code should be reusable for similar sensing deployments.

Domain Specificity: Paradigms, syntax, and abstractions should match the problem domain to facilitate natural and efficient expression of problem solutions.

Readability: Provide a syntax that promotes readability, captures domain knowledge, and is understandable to those in the domain for which the language is designed.

Concision: Language constructs should allow compact expression of common needs.

Modularity: It should be possible to add, remove, and replace pieces of the configuration with minimal "tangling".

Expressiveness: The system should allow convenient and straightforward implementation of common needs.

Extensibility: A language should interface with functionality written in other languages, or user-contributed elements.

Separation of Interface from Implementation: Language features should not force the user to grapple with the details of their implementation.

Metalinguistic Abstraction: Where possible, allow tailoring of language syntax to conveniently express required functionality.

3.3 Platform Requirements

We have developed a configuration language named Seddacco (**SE**lf **D**ocumenting **D**ata **A**nalysis and **C**ollection **CO**de) to address these requirements. Seddacco interfaces with the iEpi data collection system [1][2][3][4] to implement domain-specific features and tailors its syntax to succinctly interact with said system's abstractions. For convenience, we provide an overview of the system and its abstractions here; further information on the architecture can be found in [3] (currently in press); information on capabilities and use cases can be found in [1][2][4][33].

3.3.1 iEpi System Overview

iEpi was designed to meet the needs defined in Section 3 and can be easily configured for various data acquisition tasks. The system consists of several concurrent data processing pipelines. Each pipeline is composed of a series of tasks that either generate or manipulate information produced by tasks earlier in the pipeline. Exactly how many pipelines are produced and what tasks compose them are determined by the goals of the researcher. For example, a simple monitoring deployment might require a single GPS collection and filtering pipeline; a complex activity monitoring deployment might require many pipelines that manipulate accelerometer, heart rate, and survey data. These pipelines process information on schedules specified by the researcher, and can optionally be restricted by conditional logic to ensure more precise execution times and/or contextual triggered behavior. All data is eventually funneled to a centralized data processing service that performs actions such as encryption and compression.

3.3.2 iEpi Abstractions

Task & Parameters: Tasks are information processors to be executed. They vary in purpose, ranging from simple operations, such as accelerometer data collection, to complex data processing steps. Tasks can be augmented by developer-defined parameters, to represent values such as survey names and sensing thresholds.

Conditions: Conditions are blocks of logic that evaluate to true or false. They can be combined to form Boolean expressions and capture complex predicates that control task execution.

Task Chains: Tasks can be chained to accomplish larger data processing goals. For example [3], the task chain representing *visually map movement behavior* might consist of the tasks *sample GPS data, filter unusable results, apply Kalman filter, snap to map paths, impute gaps in location data by assuming shortest path travel,* and *save data in Google Earth format*. Task chains allow new logic to be constructed without requiring new tasks. All tasks in the system are incorporated into task chains. Single tasks are abstracted as one element chains.

Schedules: Task chains execute according to developer-defined schedules. The default schedule in iEpi is a duty cycle (e.g., measure for 30 s every 2 min), but other schedules are possible.

Implicit Task Chain: Most generated data will need to be processed. The implicit task chain is a set of tasks that is applied to all collected data unless otherwise specified; the chain provides "default" processing, including compression and encryption.

3.3.3 iEpi Requirements

Seddacco's dependency on iEpi motivates specific language constructs and compiler features:

Task Invocation: Seddacco must allow the user to invoke arbitrary tasks.

Task Parameterization: Seddacco must allow tasks to receive parameters.

Task Interoperability: Seddacco must permit the task chaining for larger data processing constructs.

Startup Logic: Seddacco must facilitate initialization.

Implicit Processing: Seddacco must relieve the user of the need to specify implicit data processing tasks, such as encryption.

Abstraction Linkage: Seddacco must allow users to combine iEpi abstractions, and should automate as much as possible [21].

These platform-specific requirements fit into our domain requirement set. They therefore serve only as guides for the language constructs to be implemented: they neither inherently hinder nor restrict the design.

4. SEDDACCO LANGUAGE DESIGN

Seddacco is an extensible domain-specific language (DSL) [21][22] designed to configure individual-centric mass data collection systems. The language is a front end for data collection systems. In accordance with widespread practice in many DSLs, the language achieves broader understandability by use of a declarative style, i.e., one that allows the user to specify *what* configuration they are seeking, rather than *how* it is implemented. Data manipulation algorithms, execution constraints, and temporal constraints are declared by the programmer, which the underlying data collection system then links to executable code. Guided by DSL and macroprogramming best practices, Seddacco's syntax, examples of which can be found in Section 6, is designed to make this process simple, natural to express, and capture domain knowledge.

4.1 Language Overview

A major goal of Seddacco is to allow for the rapid establishment and modification of device behavior based on existing functionality. The language acts as a front-end to programmer-provided constructs, such as algorithms to be performed and the schedules on which they run, that are manipulated via English-like statements that can be customized and extended on a perapplication basis. The language provides connectivity between these constructs and allows them to be recombined for use in a variety of application-specific arrangements and scenarios. Seddacco follows a design philosophy much like that of the Unix command line: programmers develop small, modular programs in the language of their choice, encapsulate them in a file or command, and then combine them using system-provided facilities. The major divergence is that where command line systems aim to be terse at the expense of readability, Seddacco aims to be verbose to encourage self-documenting and human-readable code. Specifically, Seddacco exploits the metalinguistic abstraction – a principle supporting the rich LISP family of DSLs [28] – to allow application teams to customize much of the language's syntax to their domain.

4.2 Syntax Overview

Seddacco facilitates the configuration of iEpi and allows interaction with the system's abstractions. The language's declarative approach lessens the need to understand the "plumbing" underlying these abstractions, fulfilling both platform and domain requirements. Seddacco's declarations are clause-based. Each clause is tied to a specific abstraction: for example, the Task Clause is used to specify which tasks are created; the Schedule Clause is used to specify the schedules on which these tasks will run. This allows the user to operate in terms of simple concepts, while the capacity to expressively invoke the platform's abstractions allows for succinct fulfillment of many use cases. The clauses consist of minimally delimited English-like phrases to help express domain knowledge and promote readability. The various clauses can be found in Table 1.

iEpi's scheduling distinction between initialization and lifetime operations is reflected in a Seddacco program's two main sections: a startup block and a program body. The startup block contains tasks to be executed before the remainder of the program begins, which includes configuration of implicit processing operations such as encryption and anonymization. The program body meets iEpi's requirement that all constructs can be linked, invoked, and scheduled. The standard task syntax is as follows:

Startup Block: <*task clause*> [*condition clause*] [*parameter clause*] [*chained tasks clause*] ;

Program Body: <*task clause*> <*schedule clause*> [*condition clause*] [*parameter clause*] [*chained tasks clause*] ;

Chained Tasks Clause: <*task clause*> [*parameter clause*] [*chained tasks clause*]

4.3 Linking Seddacco to Java Constructs

Seddacco acts as a front end to Java classes that implement its schedule, task, and condition interfaces. Its current version is designed to hide all non-essential programming details from the user; as such, Seddacco does not support syntax to specify a link between its constructs (e.g., *upload data*) to their underlying Java classes (e.g., *DataUploadingTask*). Exploiting the principles of separation of interface from implementation [29] and (in a circumscribed form) metalinguistic abstraction [28], Java classes instead declare *token patterns* that are matched to Seddacco statements at runtime. Matching is performed when iEpi begins and the Seddacco program is parsed. Matching is performed by comparing sequences of tokens reported by the compiler against declared token patterns made of the following elements:

Static Word: Alphanumeric characters not beginning with a number that must be present as specified e.g., the text "upload".

Variable Word: Alphanumeric characters not beginning with a number not restricted to a static character pattern. These constructs get and validate their text values at instantiation.

Number: Any number. Numbers can be qualified, i.e., they can match a string pattern (e.g., '1.3e-5') or be in a range (e.g., [0,1]).

Operator: Commonly used non-alphanumeric mathematical, monetary, and programming symbols, e.g., <=, ^, %, $.

String List: A comma delimited collection of strings, where each is delimited by quotes, e.g., "string1", "string two", "string_3!". Typically used when free-form values (e.g. URLs) are required.

Bracketed Item List: A comma delimited collection of groups of characters, where each is delimited by brackets, e.g., [item 1], [item2], [item 3]. These support within-clause parameter lists.

In accordance with the Chain-of-Responsibility pattern [30], multiple constructs may reference the same token pattern providing an analogue to method overloading. For example, suppose that a task with the token pattern *collect <sensor> data exists*, e.g., *collect accelerometer data*. A default implementation of this task might encompass all Android sensors in the API. When adding new functionality for a heart rate sensor, for example, one could add a new task that accepted the argument *heartrate*, e.g., *collect heartrate data*. Employing this pattern helps satisfy our extensibility requirement.

When two or more constructs share the same token pattern, a parsing conflict occurs. The conflict is resolved by attempting to instantiate each matching construct in the order they were provided to Seddacco's construct parser: if exactly one construct can be instantiated, compilation proceeds unhindered; if none or more than one construct can be instantiated, an error is thrown.

Token patterns are similar to function headers in other languages but provide an extra degree of lexical flexibility: in Seddacco, parameters are not restricted to a set location, which allows parameters to appear mid-statement. For example, the pseudocode function *collectData(bloodOxygen)* can be expressed in Seddacco as *collect blood oxygen data*. Seddacco further allows optional parameters to be specified in a Parameter Clause at the end of a statement, meeting our readability, knowledge expression, and domain-appropriate abstractions, and syntax requirements.

4.4 Parsing Seddacco

A Seddacco program begins execution with the compilation of its configuration file, which is initiated by breaking the document into a startup block and program body. Each block is handled in a similar fashion: a list of task chain declarations are acquired from the block via Seddacco's ANTLR-based parser. The tokens of each Task, Parameter, and Condition Clauses within these declarations are then passed to an entity known as a construct parser. Construct parsers consist of ordered trees of sub-parsers following the Tree-of-Responsibility pattern (similar to Chain-of-Responsibility) [30]. This approach was used to facilitate eventual plug-and-play functionality for extensibility: the root construct parser can take in any sub-parser provided by new modules, allowing new conditions, parameters, and tasks to be parsed, which has been demonstrated to confer great benefits in the domain-specific language context [28]. The root construct parser is instantiated, given to the Seddacco subsystem, and then used to parse the system's configuration file when iEpi starts or a task requests that the file be re-read. As each program statement is read by the ANTLR parser and broken into clauses, the root construct parser is given each clause and its associated token sequence. The parser and its subordinates then match these sequences against known token patterns for the appropriate abstraction type.

Table 1: The clauses that are used to construct Seddacco statements.

Component	Description	Examples

Task Clause	High-level statement that is used to create a task.	`collect accelerometer data`
		`issue "activity" survey`
		`upload data`

Schedule Clause	Defines the schedule on which a task chain is invoked. Can specify a burst length that tells the process's task for how long it should collect data. Begins with *every*.	`every 10 minutes`
		`every 5 minutes for 30 seconds`

Condition clause	Allows the programmer to specify when a task chain is allowed to operate. To visually and syntactically delineate conditions, they are surrounded by brackets. The beginning of a condition clause begins with *only if*.	`only if [outside]`
		`only if [outside] and [moving]`
		`only if ([outside] and [moving]) or not [raining]`

Parameter Clause	Specifies optional task parameters in a comma-separated list. Begins with the keyword *with*.	`with [recency threshold = 10 minutes]`
		`with [90 % required confidence]`

Chained Task Clause	Allows tasks to be chained together. Specified with *then*.	`task_1 then task_2 … then task_n`

5. MEETING THE SPECIFICATION

In Section 3, we detailed six requirements that Seddacco must meet to interface with its target platform iEpi, three requirements that it must meet to be a language appropriate for the use case realized through the iEpi architecture.

5.1 Platform Requirements

Seddacco meets its platform's requirements through the use of clauses, a declarative syntax, and a parser that automatically links constructs as a program is processed:

Task Invocation: Allowing tasks to be invoked using Task and Chained Tasks Clauses meets the invocation requirement.

Task Parameterization: Token patterns allow parameters to be specified mid-statement; the Parameter Clause allows awkward or optional parameters to be specified after a task has been declared.

Task Interoperability: Tasks were to be chained; this is done through the Chained Tasks Clause.

Startup Logic: A startup block exists to prepare the system prior to normal operation, which facilitates startup logic.

Implicit Processing: Seddacco's parser links all task chains to iEpi's implicit task chain, relieving the user of the need to specify implicit data processing tasks.

Abstraction Linkage: Seddacco does not require the user to link tasks, schedules, conditions, nor parameters. They are automatically joined when processed by Seddacco's parser.

5.2 Domain Requirements

Through the various clauses, Seddacco meets the three items required for it to be appropriate for the domain:

System Extensibility: Seddacco allows a variety of data manipulation algorithms to be run using Task and Chained Tasks Clauses. The tasks in these clauses encapsulate algorithms both internal to the target platform and facilitated by external applications or hardware. The unified syntax serves an important abstractional role: the user does not have to use special constructs to interact with external entities, obviating the need to distinguish between the target platform and its accessories.

Execution Specificity: Task chain execution can be restricted via the Condition Clause and modified via the Parameter Clause and/or parameters found in the Task Clause. This allows the user to tailor tasks to the context in which they are used, as required.

Temporal Specificity: All task chains outside of the startup block run on schedules; tasks in the startup block implicitly run on a schedule that runs exactly once. The Condition Clause further refines when task chains operate as it short-circuits task execution if conditions are inappropriate for execution. This allows the user to tailor the execution times of tasks, as required.

Seddacco also meets many of the optional goals:

Reusability: Seddacco promotes reusability in the same way that Unix command line systems do through pipeline-like task chains.

Domain Specificity: The user's data collection and processing needs are declared in a high-level, English-like language that captures the essential features of the problem domain but delegates execution responsibility to the underlying system.

Readability: Seddacco employs an English-like syntax, allowing natural expression of domain concepts.

Concision: Seddacco employs a clause-based syntax, which allows for concise expression of concepts such as scheduling.

Modularity: Tasks, conditions, and schedules are independent of one another, which allows them to be added and removed without altering other, operationally independent system components.

Expressiveness: Seddacco relies heavily on modules provided by the underlying system. These modules can be reused, reducing the work required to implement new operations.

Extensibility: Tasks encapsulate internal and external algorithms, allowing the language to interface with other tools and languages.

Separation of Interface from Implementation: Seddacco is a front end system, which literally makes it an interface to an underlying implementation.

Metalinguistic Abstraction: Seddacco's token patterns and custom parsers allow teams studying health behavior to readily and modularly customize the language to their application needs.

5.3 Fundamental Use Case

Activity levels, service detection, spatial analysis, and data uploads can be performed by tasks, which are invoked via Task Clauses; new data streams, both internal and external, can be facilitated by the addition of new tasks. Environment-specific sampling can be implemented by restricting the operation of task chains to the environment for which they are best suited; this is accomplished by tying conditional logic to task chains via the Condition Clause. Intelligent resource usage can be accomplished through scheduling, conditional execution, and task modification by combining one or more of the Schedule Clause, Condition Clause, and parameters (e.g., maximum number of data points to collect in each sample). The Schedule Clause further allows both near-continuous and discrete operations to be managed as its duty cycle interval and duration can be changed. Data processing can be configured and managed via Chained Task Clauses.

```
collect gps data
        every 2 minutes for 100 seconds
then update user location on map;

collect wifi data
        every 2 minutes for 100 seconds
        only if
                [gps signal unavailable within last 30 seconds]
                or not [gps signal accurate to <= 100 meters]
then localize via skyhook
        with [skyhook url "www.example.com/SkyhookQuery.php"]
then update user location on map;
```

Figure 1: Seddacco Example – Localization

```
# Startup tasks configure encryption and user anonymization.
before tasks start:
        use custom rsa encryption with [modulus 5555], [exponent 5555];
        use identifier protection with [hash value "magical hash value"];
done;

# SHED3 [4] sampled relevant sensors every two minutes for 30 seconds each.
collect accelerometer data every 2 minutes for 30 seconds;
collect magnetometer data every 2 minutes for 30 seconds;
collect bluetooth data every 2 minutes for 30 seconds;
collect compass data every 2 minutes for 30 seconds;
collect wifi data every 2 minutes for 30 seconds;

# Collect 10 battery samples periodically to determine if the phone is plugged in.
collect battery data every 2 minutes for 30 seconds with [record cap 10];

# Update the database in a resource-efficient way.
store volatile data every 2 minutes;

# Upload data to the server frequently when plugged in and infrequently when not.
upload to "http://www.example.com/shed3/collect.php" every 4 hours;
upload to "http://www.example.com/shed3/collect.php" every 10 minutes only if [plugged in];

# Perform configuration updates periodically.
update using "http:///www.example.com/shed3/updates/update.txt" every 90 minutes;

# Show a survey two to three times every day between 8:00am and 6:00pm.
issue "midday-survey.xml"
        every 2 minutes
        when [hour between 8 and 18]
        with [launch probability 0.0083];
```

Figure 2: Seddacco Example – Standard iEpi Data Collection Configuration from Existing Deployment [4]

```
collect wifi data
        every minute for 45 seconds
        only if [indoors] and [map open]
then remove unreliable wifi data
        with [-80 rssi threshold]
then trilaterate from "wifi-router-locations.csv"
then apply kalman filter
then snap to map routes
then update map with calculated position;
```

Figure 3: Seddacco Example – Complex On-Device Data Processing

6. EXAMPLES

We present the syntax of Seddacco as it might be given in a three scenarios: a canonical [15] ad-hoc localization task; a simple iEpi-style mass health data collection experiment; and a complex data collection scenario. In most examples the startup block is omitted as these normally contain rudimentary setup functions.

Ad-hoc Localization: We begin with a hypothetical set of statements for on-device localization. In Figure 1, localization is performed periodically by collecting GPS and WiFi data. When GPS data is difficult to acquire – e.g., when the user is indoors – WiFi data is used to generate position using a trilateration service connected in the WiFi task chain. When data is collected, it is output for storage via the implicit task chain and used to update a map.

Simple Mass Data Collection: iEpi was originally conceived for mass data collection. Figure 2 depicts the Seddacco configuration used in the SHED3/SHED4 [4] experiments updated to match the language's current syntax.

Complex On-Device Processing: Many traditional sensor network applications collect and apply operations to data off-device, often due to limited processing and battery power. With modern smartphones this limitation is gone. Figure 3 depicts a hypothetical Seddacco configuration for a complex on-device data manipulation and collection task.

7. DISCUSSION

Seddacco is unique in that it provides a combination of functionality that many macroprogramming languages avoid. This is most obvious when comparing Seddacco to language classification systems [23][25]. Bai's classification system [23] indicates that Seddacco supports mobile, periodically sampling applications with periodic data transmission mechanisms. The language allows for actuation via Java statements and conditions. The language provides restricted support for interpreting data in-network, but does not permit data aggregation within-network as Seddacco is designed for devices at least intermittently connected to a wireless backbone. Motolla's [25] classification indicates that Seddacco supports sense-and-react applications with a many-to-one interaction pattern. Nodes are assumed to be mobile with regional interests as opposed to global interests. Data processing is periodic. The authors are not aware of other behavioral sensing macroprogramming languages that provide this feature set.

From a language perspective, Seddacco's use of metalinguistic abstraction sets it apart from other data collection configuration languages, not only in terms of flexibility, but also in terms of the degree to which it can be understood across interdisciplinary research teams. While the other languages benefit from metalinguistic abstraction, we are unaware of any that use that capacity to benefit non-technical end users. Seddacco's innovative means of language extension further raises the intriguing potential for declarative syntax customization by non-technical end-users.

Seddacco is an English-like front end to the larger iEpi system. This approach supports simple specification of data processing pipelines to perform more complex operations. In principle, this would seem to make the language easy to scan for high-level errors and might make it easier to read and comprehend than other languages. However, linking Java constructs to Seddacco statements is a slow process, although significantly simpler than connecting and manipulating constructs manually.

The evolution of Seddacco to date has been driven by the need to improve the usability of the iEpi system [1][2][3][4]. Thought has gone into supporting more interesting constructs to reverse this relationship and instead drive the evolution of iEpi. Many languages have event support. Despite the fact that events may be difficult for non-experts to utilize [24], there are situations where they are useful. Extending both the architecture and language to support events would be desirable. iEpi supports rudimentary task trees, i.e., the output of tasks can be piped to multiple output sources; Seddacco could be augmented to allow easy access to this currently hidden feature. Declarative specification of token patterns associated with Seddacco constructs could greatly improve linking and – more intriguingly – allow non-technical users to adapt the language without recourse to software development expertise. Because Seddacco is similar to Unix command shells, it could be ported to non-smartphone environments to broaden its scope of use.

8. CONCLUSION

Measuring human behavior using smartphones is a challenging exercise demanding expertise spanning multiple disciplines. Formulating a language that addresses the diverse needs of health studies in the health sphere is a daunting task. We have elucidated the general requirements for a system to do so, covering the health use case, the constraints of Android, and considering the needs of diverse interdisciplinary teams typically involved in digital health behavior studies. From these requirements, we were able to create the highly flexible, modular, expressive, and broadly accessible configuration language, Seddacco, that has been demonstrated to work by facilitating a number of published deployments [1][2][3][4]. As a language, Seddacco is further distinguished by its application of metalinguistic abstraction via syntactic and semantic extensibility, which both supports remarkable level of inter-application flexibility and speaks to key needs of interdisciplinary teams.

9. REFERENCES

[1] Hashemian, M. S., Stanley, K. G., Knowles, D. L., Calver, J., and Osgood, N. D. 2012. Human network data collection in the wild: The epidemiological utility of micro-contact and location data. *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, ser. IHI '12. New York, NY, USA: ACM, pp. 255–264

[2] Hashemian, M., Knowles, D., Calver, J., Qian, W., Bullock, M. C., Bell, S., Mandryk, R. L., Osgood, N., and Stanley, K. G. 2012. iEpi: An end to end solution for collecting, conditioning and utilizing epidemiologically relevant data. *Proceedings of the 2Nd ACM International Workshop on Pervasive Wireless Healthcare,* ser. MobileHealth '12. New York, NY, USA: ACM, pp. 3–8

[3] Knowles, D.L., Stanley, K.g., Osgood, N.D. A Field-Validated Architecture for the Collection of Health-Relevant Behavioural Data. *ICHI 2014 (Accepted).*

[4] Petrenko, A., Bell, S., Stanley, K., Qian, W., Sizo, A., and Knowles, D. 2013. Human spatial behavior, sensor informatics, and disaggregate data. *Spatial Information Theory, ser. Lecture Notes in Computer Science*. Springer International Publishing, vol. 8116, pp. 224–242

[5] Hnat, T. W., Sookoor, T. I., Hooimeijer, P., Weimer, W., & Whitehouse, K. 2008. Macrolab: a vector-based macroprogramming framework for cyber-physical systems.

Proceedings of the 6th ACM conference on Embedded network sensor systems (pp. 225-238). ACM.

[6] Whitehouse, K., Zhao, F., & Liu, J. 2006. Semantic streams: A framework for composable semantic interpretation of sensor data. *Wireless Sensor Networks* (pp. 5-20). Springer Berlin, Heidelberg.

[7] Min, C., Yoo, C., Lee, Y., Song, J. 2011. Healthopia: Towards your well-being in everyday life. *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, ser. ISABEL '11. New York, NY, USA: ACM, pp. 108:1–108:5.

[8] Aanensen, D. M., Huntley, D. M., Feil, E. J., al Own, F., Spratt, B. G. Epicollect: Linking smartphones to web applications for epidemiology, ecology and community data collection. *PLoS ONE*, vol. 4, no. 9, p. e6968, 09 2009.

[9] Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., & Culler, D. 2003. The nesC language: A holistic approach to networked embedded systems. *Acm Sigplan Notices* (Vol. 38, No. 5, pp. 1-11). ACM.

[10] Levis, P. (2004). The TinyScript language. A Reference Manual.

[11] Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W. 2002. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *OSDI*

[12] Madden , S., Franklin, M., Hellerstein, J., Hong , W. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Systems*.

[13] Welsh, M., & Mainland, G. 2004. Programming Sensor Networks Using Abstract Regions. *NSDI* Vol. 4

[14] Whitehouse, K., Sharp, C., Brewer, E., & Culler, D. 2004. Hood: a neighborhood abstraction for sensor networks. *Proceedings of the 2nd international conference on Mobile systems, applications, and services* (pp. 99-110).

[15] Gummadi, R., Gnawali, O., & Govindan, R. 2005. Macro-programming wireless sensor networks using kairos. In Distributed Computing in Sensor Systems (pp. 126-140). Springer Berlin Heidelberg.

[16] Luo, L., Abdelzaher, T. F., He, T., & Stankovic, J. A. 2006. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems* (TECS), 5(3), 543-576.

[17] Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., Wood, A. 2004. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *Proceedings. 24th International Conference on Distributed Computing Systems* (pp. 582-589). IEEE.

[18] Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. 2005. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. *Proceedings of the 2005 Workshop on End-to-end, Sense-and-respond Systems, Applications and Services*, ser. EESR '05. Berkeley, CA, USA: USENIX Association, pp. 19–24.

[19] Newton, R., Morrisett, G., & Welsh, M. 2007. The regiment macroprogramming system. *Proceedings of the 6th international conference on Information processing in sensor networks* (pp. 489-498). ACM.

[20] Kang, S., Lee, J., Jang, H., Lee, Y., Park, S., & Song, J. 2010. A scalable and energy-efficient context monitoring framework for mobile personal sensor networks. *IEEE Transactions on Mobile Computing*, 9(5), 686-702.

[21] Mernik, M., Heering, J., & Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4), 316-344.

[22] Spinellis, D. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1)

[23] Bai, L. S., Dick, R. P., & Dinda, P. A. 2009. Archetype-based design: Sensor network programming for application experts, not just programming experts. *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks* (pp. 85-96). IEEE Computer Society.

[24] Miller, J. S., Dinda, P. A., & Dick, R. P. 2009. Evaluating a basic approach to sensor network node programming. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (pp. 155-168). ACM.

[25] Mottola, L., & Picco, G. P. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*,*43*(3), 19.

[26] Wild, C.P. 2012 The exposome: from concept to utility. *International journal of epidemiology* 41.1: 24-32.

[27] Louis, B., Germaine, M., & Sundaram, R. 2012. Exposome: time for transformative research. *Statistics in medicine*, *31*(22), 2569-2575. Volume 31, Issue 22, pages 2569–2575, 28

[28] Abelson, H., Halfant, M., Katzenelson, J., & Sussman, G. J. 1988. The LISP experience. *Annual review of computer science*, *3*(1), 167-195.

[29] Liskov, B., and Guttag, J. 2000. Program development in JAVA: abstraction, specification, and object-oriented design. Pearson Education.

[30] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1994. Design patterns: elements of reusable object-oriented software. Pearson Education.

[31] Aharony, N., Gardner, Alan., and Sumter, C. FunF: Open Sensing Framework. Online: retrieved June 2014. http://www.funf.org/about.html

[32] Chronos Mobile Technologies Inc. Chronos. Online: retrieved June 2014. https://www.getchronos.com/

[33] Qian, W., Stanley, K. G., & Osgood, N. D. 2013. The impact of spatial resolution and representation on human mobility predictability. *Web and Wireless Geographical Information Systems* (pp. 25-40). Springer Berlin Heidelberg.

[34] Pathak, A., Mottola, L., Bakshi, A., Prasanna, V. K., & Picco, G. P. 2007. Expressing sensor network interaction patterns using data-driven macroprogramming. *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, ser. PerCom Workshops' 07. (pp. 255-260). IEEE.