The Design and Use of a Smartphone Data Collection Tool and Accompanying Configuration Language

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Dylan Knowles

Permission to Use

Abstract

Understanding human behaviour is key to understanding the spread of epidemics, habit dispersion, and the efficacy of health interventions. Investigation into the patterns of and drivers for human behaviour has often been facilitated by paper tools such as surveys, journals, and diaries. These tools have drawbacks in that they can be forgotten, go unfilled, and depend on often unreliable human memories. Researcher-driven data collection mechanisms, such as interviews and direct observation, alleviate some of these problems while introducing others, such as bias and observer effects. In response to this, technological means such as special-purpose data collection hardware, wireless sensor networks, and apps for smart devices have been built to collect behavioural data. These technologies further reduce the problems experienced by more traditional behavioural research tools, but often experience problems of reliability, generality, extensibility, and ease of configuration.

This document details the construction of a smartphone-based app designed to collect data on human behaviour such that the difficulties of traditional tools are alleviated while still addressing the problems faced by modern supplemental technology. I describe the app's main data collection engine and its construction, architecture, reliability, generality, and extensibility, as well as the programming language developed to configure it and its feature set. To demonstrate the utility of the tool and its configuration language, I describe how they have been used to collect data in the field. Specifically, eleven case studies are presented in which the tool's architecture, flexibility, generality, extensibility, modularity, and ease of configuration have been exploited to facilitate a variety of behavioural monitoring endeavours. I further explain how the engine performs data collection, the major abstractions it employs, how its design and the development techniques used ensure ongoing reliability, and how the engine and its configuration language could be extended in the future to facilitate a greater range of experiments that require behavioural data to be collected. Finally, features and modules of the engine's encompassing system, iEpi, are presented that have not otherwise been documented to give the reader an understanding of where the work fits into the larger data collection and processing endeavour that spawned it.

Acknowledgements

There are an enormous number of people to thank in the production of this work. I thank Eva Anderson, Roland Dyck, Farjana Eishita, Amin Tavassolian, Erik Volz, Heather Ward, and Alexander Yaholnitsky for descriptions of their iEpi-related work, their help in shaping the evolution of the system, and their extraordinary patience and assistance in squashing nasty bugs. I would like to thank the Natural Sciences and Engineering Research Council of Canada and the Graphics, Animation and New Media Network Centre of Excellence for funding the development of the architecture. I would like to thank Mohammad Hashemian for his construction and simplification of parts of iEpi, and for his guidance in non-academic areas relevant to the system. I would like to thank Kurt Krueger, Winchell Qian, and William Van der Kamp for their use of and contributions to the system, as well as their extreme patience in dealing with me during a long and stressful period of my masters thesis. I would like to thank Christopher Dutchyn for his advice in limiting the scope of the programming language that powers this work; I would likely still be building it had he not reminded me that I was not building production level software. I would like to thank our tech staff – Merlin Hansen, Cary Bernath, Greg Oster, and and Seth Shacter – for making sure I had the resources to get my work done. (The support of the entire computer science department was pretty indispensable in this regard, too.) Mark Eramian is also to be thanked for his godlike LaTeX knowledge and the countless times he has come to my rescue during the writing process. There are external collaborators I would like to thank for their patience and the fantastic experiences I have had in working with them, but must refrain from doing to avoid breaching confidentiality. In light of these acknowledgements, it is further worth noting that although stylistic convention dictates the use of "I" in thesis documents, much of the work contained herein – specifically study execution and post-hoc data analyses – would not have been possible without a "we."

Of course, no acknowledgement section would be complete without mentioning my supervisors and parents. My supervisors, Nathaniel Osgood and Kevin Stanley, allowed me to conduct this work for four years, have put me in touch with an enormous number of wonderful collaborators, have provided me with invaluable guidance, and have made possible a large number of academic and non-academic experiences. They have given me a great deal of freedom in conducting my work, constructing the system, and extending it in ways I saw most valuable; considering our group now has a massive, robust, and flexible system being clamored for by collaborators all over the world, that strategy seems to have worked out. My parents, Cliff Knowles and Judy Knowles, have been incredibly supportive during this entire process. I cannot imagine getting through my degrees and the many stressful periods that accompanied them without their support, care, and love.

Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard
AGPS	Assisted GPS
API	Application programming interface
App	Smartphone App
Celltower	Cellular Network Tower
CMQ	Context Monitoring Query
DAG	Directed Acyclic Graph
DSL	Domain Specific Language
ECB	Electronic Codebook
EEG	Electroencephalogram
FC	Functional Components
GPL	General Programming Language
GUI	Graphical User Interface
IMDCS	Individual-Centric Mass Data Collection Systems
JSON	JavaScript Object Notation
MAC	Media Access Control
RFID	Radio-Frequency Identification
RSA Encryption	Ron Rivest, Adi Shamir and Leonard Adleman Encryption
SDK	Software Development Kit
SHED	Saskatchewan Human Ethology Dataset
STEPS	Sensorium Typed Execution Plans
XML	Extensible Markup Language

Chapter 1

Introduction

The study and notion of population health – the complete range of factors that determine health and affect the entire population [103] – has ranged in approach and definition over the centuries. In the western world, the medical community has increasingly sought to enhance the health of the population via evidence-based methodologies [19]. This endeavour has transformed simpler remedies of the past with increasingly sophisticated understanding, leading to the eradication of deadly diseases, severe reductions in infant mortality rates, and extended lifespans. The techniques to achieve these gains are increasingly methodical and sophisticated. However, there remains many important gaps. One of the most important relates to the study of health-related behaviours outside of a clinical setting – behaviours which serve as an important mediator for most efforts to improve the health of the population. Tools in this domain, such as surveys and activity diaries, tend to shift the burden of data collection to the participant. As participants are not always diligent, unbiased, nor accurate data recorders, this can impose severe quality limitations. Health professionals relying on information from these tools are therefore left in an unfortunate position: given potential quality problems in data related to human behaviour, how much trust can be placed in it, and how sure can they be in making decisions that will impact the long-term health of individuals?

Well-known methods for observing and recording human behaviour such as surveys, take-home diaries, direct observation, and interviews can capture participants' activities, their perception of the importance of these activities, and can be used to inform future behavioural studies. These methods are also plagued with problems. Surveys are limited in the information that they can acquire and, as participant self reporting is subjective, can make it difficult to design surveys that consistently quantify behaviour across participants. Survey answer quality and completion rates can also be limited by the patience, memory, and abilities of the participant [49]: if life is perceived as too busy, or a survey is simply forgotten, a survey will go unanswered. Human memory and perception is often biased, causing participants and researchers alike to emphasize some events at the expense of others. Social and cultural pressures further act to impede research quality as they can cause participants to avoid reporting embarrassing behaviour, skew answers to match social norms, or simply frame their responses to please the researcher [79]. Similar problems can be encountered with diaries, observation, and interviews. The latter two are good at soliciting and precisely recording interesting phenomena, but – in the absence of a skilled and trusted interviewer or observer – may alter participant behaviour or cause participants to withhold sensitive but critical information. All of these limitations impact data quality and, in turn, evidence-based decision-making related to population health.

Advances in portable technologies may help reduce observation and bias difficulties encountered by researchers studying health behaviour. Devices such as step counters, GPS devices, heart rate monitors, and electronic blood glucose meters can all play a part in improving data quality as they can provide fine-grained, quantitative, and standardized channels through which data may be acquired. These are important factors when attempting to quantify human behaviour as they can simplify the comparison between bodies of evidence gathered in behavioural research and can provide data that is less subject to participant biases. These solutions are often disconnected by both physical and software restrictions, making it potentially difficult to combine their data within a study at the level of an individual – a key factor for many types of health insight.

More encompassing technological solutions have been proposed to monitor individuals, areas, and environments. Many of these technologies, known as *wireless sensor networks*, have seen significant use in non-human monitoring applications: they are good at observing moving entities, reporting data in real-time, can have significant deployment lifespan, and can be deployed in remote environments to silently study phenomena of interest [28, 67, 99]. Unfortunately, they can also be easily damaged during a participant's daily activities, may be vulnerable to static buildup if worn on a participant's body, can be bulky or uncomfortable to wear, and can be difficult to program and deploy correctly [40, 49]. Many of these systems are difficult to update for new deployments, creating entry barriers to use by new researchers or researchers outside the wireless sensor network community. Some solutions even act as unintentional social markers, e.g., ankle-mounted collection devices that are similar in appearance to those that monitor the movements of high-risk criminal or sexual offenders.

Less fine-grained approaches have been developed to circumvent these problems. In particular, cellular network tower (henceforth, celltower) records have been used to identify human mobility patterns and other human behaviours [23, 35, 86, 97, 104]: because each celltower's location is known, the location of an individual communicating with a tower can be approximated when a call or text message is sent; this, in turn, can be used to infer or estimate behaviour and other personal information, e.g., an individual's exposure to environmental pollution or socioeconomic conditions based on the location in which their phone is most often used. A number of researchers have taken this approach [23, 35, 86, 97, 104] and their work has improved understanding of human movement, patterns, and predictability. These methods, however, present privacy and ethical concerns: most studies do not state if they requested or were required to disclose to participants that they were using such individuals' call behaviour. Given rising awareness of privacy issues and new laws surrounding data usage, these kinds of studies may prove increasingly difficult to initiate and conduct. The spatial and temporal resolution of these methods also present difficulties in assessing fine-grained behavioural factors. For example, it is not possible to collect reliable activity data from these records, nor is it possible to reliably identify the person-person and person-environment interactions needed to accurately trace the movements of a disease through a population.

With the advent of powerful mobile consumer electronics such as smartphones and smartwatches and the programming platforms they provide, many of these problems could be reduced or largely eliminated. Sensors normally found only in specialized devices, such as pulse monitors and the accelerometers found in pedometers, are now integrated directly within smartphones. This opens a world of behavioural and health monitoring opportunities. Sensor fusion can take place both on- and off-device, allowing relevant information to be combined to detect participant context. For example, high accelerometer use, accurate GPS signal, high temperature, and the month of June in southern Canada might indicate outdoor activity; proximity to rural celltowers could further restrict this information to outdoor farm chores. This and other sensor data can be used to trigger collection or reporting measures for even greater specificity: it is not difficult to imagine that at times of irregular travel patterns one might increase the rate at which other sensors are used, or that prolonged, increased blood pressure measures be used to trigger medical notifications. It is also possible to use sensor-derived contexts to invoke traditional tools, such as surveys and diaries, at moments of interest. Smartphone sensor data collection apps can be made to interact with external, specialized tools such as survey systems. App infrastructure also provides mechanisms for updating and distributing the code of collection apps, making deployment and device management simpler.

Smartphones reduce many of the day-to-day problems encountered by traditional pen and paper tools and custom hardware. Smartphones are frequently woven into their owners' lives and are generally carried by their owners. Smartphone-based data collection therefore has the potential to capture much information about participants' lives. Smartphone apps (henceforth, apps) can issue reminders, can launch traditional tools exactly when required, can provide sensed context such as location to survey and diarying applications, and are weightless and spaceless beyond the hardware platform on which they run. Paper surveys and diaries, in contrast, are bulky, silent, and have no knowledge of their previous use nor current spatial context. Apps have upload capabilities that can transmit data on regular intervals, which reduces the likelihood that diary and survey entries will be lost. Smartphones are built to withstand the abuse of careless human users and environmental risks (e.g., static discharge) reducing the likelihood that collected data will lost due to damage. Smartphones are normal constituents of many participants' lives [16, 82]: this makes the devices less likely to impact participant behaviour, provided that they do not cause too many interruptions, and do not unduly shorten battery lifetimes.

A significant difficulty with smartphone data collection systems is that, like many other technological instruments, they must be built and then configured for each research task. As smartphones are complex systems meant to facilitate an enormous number of application types, expertise is required in developing these systems. Related technical needs resemble that of an operating system, requiring collection scheduling, application management, resource management, database manipulation, file storage, network communication, and GUI development. This list does not even begin to cover the needs associated with potential use cases a health researcher might envision, which can range from location detection to monitoring the long-term effects of social networks on activity and food choices. The list also does not cover any concerns regarding the quality, reliability, and flexibility of these systems expected and sought by researchers.

It is wasteful to develop unique systems for each study. The costs of and the potential for software errors when developing new systems points to the utility of a broadly applicable and configurable system for collecting health and behavioural data. The literature is rife with examples of collection systems aiming to facilitate the collection of human behaviour data [23, 40, 46, 83]. This previous work provides guidance and lessons learned that the developers of a general health and behavioural data collection systems should follow.

The goal of this work is to demonstrate that it is possible to develop a flexible, extensible, modular, configurable, and versatile data collection system to facilitate and help resolve the difficulties encountered in behavioural data collection efforts. Over the past 4-5 years, I and others have developed the end-to-end iEpi data collection system [38, 39, 49, 50] to accomplish this goal. The system includes data collection applications, a data management system, specialized accompanying applications, and a limited suite of analysis tools. The work is the result of the combined efforts of nearly a dozen individuals and has many compartmentalized components. This document describes the design and deployment of this larger system's data collection engine – colloquially referred to as "the logger" – and the custom programming language, Seddacco, that is used to configure the logger for individual experiments. These components are shown to accomplish the aforementioned goals – flexibility, extensibility, modularity, configurability, and deployment versatility – by documenting several deployments in which they have been successfully employed.

iEpi is a widely used system both within our research group and abroad, having been used in completed or ongoing studies in Saskatoon, New York, New Mexico, Michigan, Massachusetts, and beyond. Given the widespread use of the system, the diverse team supporting its segregated components, and the potential absence of the author following the completion of this thesis, a detailed future work section is provided to serve as a roadmap for its future expansion.

Chapter 2
Related Work

The work of this thesis concerns the design, implementation, and deployment of a smartphone-based data collection tool, accompanying configuration language, and auxiliary data processing tools. These tools are designed to collect data on human health and behaviour. There is significant related literature on these subjects, ranging from direct studies of behaviour using simple technological aids to entire data networks being harnessed for population studies. As the focus of this thesis is not the data derived by these systems but rather the systems themselves, this review will largely address the technical aspects of related work as opposed to the behavioural monitoring implications they provide.

2.1 Sensing Behaviour: From Humble Beginnings to Mass Data Collection

A vast and diverse set of studies have been performed to better understand human behaviour using traditional methods such as surveys, diaries, and direct observation. These approaches are limited in their ability to provide fine-grained, quantitative, and objective data on human behaviour. Some researchers have begun to investigate modern sensing technologies as a means to compensate for or rectify these limitations. For example, Isaacson [84] evaluated the use of a Magellan GPS receiver and Ituran personal tracking unit to record the location of individuals. Elgethun [26] investigated the use and efficacy of tracking children using clothing-integrated GPS systems. Kamleitner [44] used a smartphone-based system that prompted users when surveys were to be completed.

More technologically involved approaches are useful when behavioural monitoring experiments grow to include larger populations or require greater scrutiny of individual behaviour. In particular, several researchers have exploited celltower, Bluetooth, WiFi, and RFID technologies to track individuals, their habits, and their movements. Of these technologies, the use of celltower data – specifically, call log data – has received the most attention. When a call or message is sent, a user's device must necessarily connect with a celltower; at this time, the identifier of that celltower is recorded. As celltowers are fixed in location and individuals generally connect to the closest celltower, celltowers can act as a proxy for the user's location. The location data drawn from celltower data can then be fused with other sources to infer other behavioural and population properties, e.g., the population density of an area extrapolated from the number of detected nearby devices.

Eagle *et al.* [23] used mobile phones to uncover patterns in human interaction. This is possible because, as Eagle notes, these devices are typically carried and used by their owners. The devices' locations, interactions, activities, and proximities to others can therefore be used as a proxy for the same variables of their owners. As such, they are ideal when studying individuals and organizations. While Eagle's approach was focussed largely on individual habits and movements, the same approach could be extended to studying human health. For example, associations might be identified between the proximity of individuals to economically poor areas and overall wellbeing, or between two individuals following similar life trajectories and their long-term health outcomes.

Eagle captured participant location via cell tower ID and Bluetooth signals emitted from other devices. Bluetooth capture and collection was facilitated through the use of BlueAware, a cellphone application, and Bluedar, a standalone sensing system connected to a wireless network. Since cell phones are energy constrained, on-device data collection was performed on a duty cycle to improve battery life. Eagle notes that during this process individuals were almost always able to connect to cell towers or see other Bluetooth-enabled devices; these facts allow phones, and their owners, to be located and studied at most periods throughout the day. As mobile coverage has improved since Eagle's original work, this fact can likely be asserted with greater confidence in modern celltower-based studies.

Gonzalez *et al.* [35] tracked 100,000 mobile phone users over six months. They used this information to better understand human mobility patterns and existing models of human mobility. Their data was drawn from call and messaging records: when a call or message is sent or received, the tower facilitating the communication is recorded and its location is used as a proxy for the user's location. They found that human behaviour exhibits high degrees of spatial and temporal regularity, a finding that – while intuitive – may have been difficult to objectively verify on a large scale without technological means.

Song *et al.* [86] used mobile phone records, which use nearby celltowers as a proxy for location, to analyze the entropy of 50,000 individuals. Their work finds that peoples' activities are up to 93% predictable, which, in conjunction with other work finding that human mobility follows a fractal pattern [52], can be used to make better models of human movements and interactions. These findings are of use in health applications wherein mobility-sensitive simulations of human behaviour must be made on the basis of intuition, approximation, or from the results of small-scale studies.

Wang *et al.* [97] analyzed the mobility patterns of six million mobile phone users. This information allowed them to better understand the link between proximity, movement similarity, and the social ties between individuals, as well as giving them access to the daily routines of individuals. Their work was facilitated by call detail records, which provide information about where calls are made based on all tower locations through which the call was routed. The data was presumably retrieved from network providers as it seems unlikely that explicit permission was drawn from six million users; data collection without informed consent might be problematic in some jurisdictions due to privacy laws, potentially limiting widespread use of this technique. Data collection of this kind also suffers from spatio-temporal weaknesses: the fewer calls and text messages an individual sends, the less accurate estimations of their behaviour becomes. This further biases data as more will be known about those who frequently use their devices and less about those who do not.

Yuan *et al.* [104] used a mobile phone dataset covering a million individuals to study human mobility patterns. They used celltower information to discretize the region from which this dataset was drawn, allowing them to more easily compare the movements of individuals. Their work, in conjunction with algorithms drawn from the fields of speech recognition and signal processing, allowed them to identify typical and outlying behaviours. This approach is useful in the context of health because it allows situations and times of interest to be pinpointed once historical data has been established.

2.2 A Finer Grained Approach to Data Collection

Two notable problems exist when acquiring and using celltower-derived data. First, data drawn from network towers is coarse grained; it is not generally possible to acquire the metre-level accuracy seen in GPS systems that may be required to assess behaviour patterns. Second, informed consent may be infeasible to achieve given the enormous populations generating the celltower data. Addressing these shortcomings may be as simple as combining data from both coarse and fine sources, such as celltower data and GPS traces, or as complex as designing completely novel, participant-downloaded collection tools to address shortcomings in existing methods. These approaches can be loosely divided based on the individualistic or collective nature of their collection methodologies. For the purposes of this review, *individualistic monitoring tools* are designed to record and report data on a specific participant for the duration of a study; examples include data collection apps placed on the participant's phone or a body-mounted camera system. In contrast, *collective monitoring tools* are designed to monitor several participants at once or qualities of an area participants frequent; examples include a wireless sensor network that attempts to track the movements of hospital staff throughout a ward, or a sensor system that identifies how consistently staff entering airport bathrooms use handwashing stations.

2.2.1 Individualistic Monitoring Tools

Individualistic monitoring tools are designed to gather data on an individual rather than the population or environment to which they belong. This is desirable when an individual's habits – and changes in these habits – are the subject of study. Individualistic monitoring tools are suited to identifying individual-level phenomena, e.g., quantifying the impact of an intervention on an individual's behaviour, observing how an individual interacts with their friend group, or directly eliciting information from an individual. They are not suited to monitoring populations at a high level or the environment in which they exist. For instance, it is conceptually simpler to monitor the usage of various resources in an office building using dedicated, distributed sensors than it is to infer this data from individualistic tools most likely covering only a subset of the population. As this thesis describes the construction of an individualistic monitoring tool, attention to the architecture, components, and configuration languages of related tools are given particular focus.

Kirovski *et al.* [48] propose a system, HealthOS, with which to unify health-related apps, technologies, and off-device applications. The system is motivated by the fact that most research applications have been developed in a closed fashion that makes interoperability difficult. HealthOS provides sensor interoperability, energy management, common user interfaces, data visualization, communication facilities, and privacy mechanisms [48]. The system is broken into three components: sensing modules, a mobile personal data hub, and a personal computer. A sensing module would provide HealthOS access to the outside world. Each module would include sensors from which data is drawn as well as data modelling, compression, analysis, and encryption services. Upon collecting data, a module would send data to the personal data hub. This hub could be a dedicated device or, more likely in modern times, a smartphone outfitted with the HealthOS SDK and framework. The data hub would further process and analyze data reported by sensor modules, provide the user with an interface to access data, provide privacy and reliability policies, and communicate data with the server. The server would provide computation-heavy services, such as modelling, analysis, sensor fusion, and medical library searches to match sensor data with medical phenomena and information. The research itself is a position paper and means that HealthOS does not exist. A robust, extensible implementation of such a system, however, would appear to be a boon to the health field.

EpiCollect [1] is a smartphone app that allows individuals to record data entries, tag them with GPS coordinates, and upload them to a server for analysis, visualization, mapping, and on-demand filtering. The system was designed to allow epidemiologists and ecologists that collect data points in the field to record and then automatically process and store data. This is in contrast to traditional data collection methods, which require the manual collection of data points, e.g., soil pH, photographs, location, etc., and their subsequent insertion into a database, that are tedious and prone to transcription errors [1]. The original system was composed of a series of interfaces on-device, a data upload mechanism, and a visualization tool. In the most recent version [56], users exploit an online form builder to specify the data fields of interest that the system then uses to generate on-device interfaces for collecting data and on-server databases for data storage. Collected data can be coupled with GPS coordinates and a variety of types of media. This architecture fills the needs of many individuals who need to enter data for discrete phenomena such as flu reports, medical clinic locations, and observed environmental hazards. Its simplicity, however, does not allow it to record the pervasive, continuous data required in many behavioural studies. It is not clear if the creators wish to extend the application in this way, which may limit its generalizability in the health field; its current abilities, however, make for a very useful tool in its target domain.

Lee *et al.* [52] create a model of human mobility named SLAW. Their model is drawn from a variety of work following humans, including GPS traces, cell phone location tracking [35], and location estimation using wireless access points. They found that their approach reproduced patterns found in human mobility not expressed in existing models. This demonstrates that merging data from a variety of sources can produce valuable results, which provides impetus for other tools to facilitate multisource and multisensor data collection.

Jigsaw [57] presents a pipeline architecture to facilitate efficient data collection and analysis on behalf of other applications. It allows client applications to request three types of data: accelerometer, magnetometer, and GPS. The system uses a pipeline architecture to break down processing tasks specific to each data type. Each sensor pipeline can decide its own sampling regime. They can also report refined information as opposed to producing raw sensor data, e.g., activity inferences rather than accelerometer data. Jigsaw's design allows for some separation of concerns within the system. System diagrams, however, make the pipelines seem tightly coupled, which creates concern regarding the system's scalability.

Celltower data is good for coarse-grained tracking of individuals, but is not appropriate for metre-level measurements. GPS can instead be used for this purpose, but does not always reliably provide this level of spatial resolution indoors nor in dense urban areas. For the purpose of precisely studying the movements of individuals indoors, Bell *et al.* [8] evaluated a Windows-based system called SaskEPS to locate individuals using WiFi access points. By creating an accurate database of WiFi access point locations, performing signal strength calibration, determining correction factors for each access point, and trilaterating the user's position based on this information, GPS-like accuracy was achieved. In the context of human monitoring, this provides the ability to study individuals both indoors and outdoors.

Min *et al.* [64] proposed and implemented a prototype of the Healthopia system. Healthopia was developed to provide smartphone apps with health-relevant data without needing to re-invent resource management, sensor processing, communication, efficient context monitoring, and privacy infrastructure. Healthopia's architecture is composed of four components, namely, the Health Monitor, Privacy Controller, Sensor Broker, and Resource Efficient Sensor Controller. The Health Monitor is designed to continually monitor sensor information and report changes in health-relevant phenomena to registered apps. The Privacy Controller allows the user to specify trusted apps that can access data and use other mechanisms provided by the Health Monitor. The Sensor Broker allows the device to discover, connect, and communicate with external wearable sensors. The Resource Efficient Sensor Controller allows for the efficient monitoring of changes in health information using Context Monitoring Queries (CMQs) [45] and resolves shared resource conflicts.

The idea of Healthopia is enticing. It also seems plausible as the underlying service infrastructure of CMQ purportedly scales well with the number of applications requesting and the number of sensors providing information [45]. It appears, however, to be restricted in the data it can provide; the authors make no mention of extensibility or code insertion points that would be required to add new health monitoring capabilities. It is not clear how the Sensor Broker would connect with newly detected sensors nor process the raw, presumably unknown data they would provide. It is also not clear how the system would facilitate non-standard requests from client apps. The authors do not report the use of Healthopia in real world studies, but rather evaluate its efficiency through the use of simulations.

Nam *et al.* [70] developed a wearable system that fused data from a belt-mounted accelerometer and pendant-like camera to analyze and classify activity levels. The authors present a system architecture to combine sensors, process data, and upload analysis results. The on-device system is composed of three layers, i.e., a communication layer, processing layer, and sensing layer. The sensing layer analyzes data from each sensor and sends it to a cache. The processing layer analyzes the cached data to classify activity. The communication layer sends the resulting dataset to a server for further processing, visualization, and utilization. The server uses a single layer with several modular components including a context mining module, life-log viewer, healthcare monitor, network communication module, context-aware middleware, and a database. These components are used to further process, visualize, present, and facilitate interaction with data with the goal of plug-and-play extensibility built into the design. It is not clear, however, how fully this ideal architecture has been implemented and if it has been developed or used beyond a prototype stage. It is also not clear if the system could be readily extended beyond its two primary sensors.

2.2.2 Collective Monitoring Tools: Wireless Sensor Networks

It is sometimes valuable to view populations as a whole rather than as a collection of individuals. In these situations, wireless sensor networks, i.e., distributed collection and reporting devices connected by wireless radios, can be useful. They are suited to observing large areas and the forces acting within them; for example, wireless sensor networks might be deployed to monitor the effectiveness of handwashing advertisements in a hospital on hand sanitizer usage. They are generally ill-suited to capturing personally intimate facets of human behaviour. For example, it is difficult to infer the activity levels of an individual through an off-body wireless sensor network; it is even more difficult to ascertain an individual's emotional state. While the contribution of this thesis is not a wireless sensor network, the architectures, languages, and themes of these works are related and therefore warrant attention.

Cattuto *et al.* [15] used a wireless sensor network of RFID devices to identify person-to-person interactions. Their framework is scalable, can identify face-to-face interactions, and identify proximity down to sub-metre levels. Their devices can be integrated into conference badges and feature a radio by which data may be communicated. This technology is useful in pinpointing social networks, which are known to be influencers of human behaviour [90] and health.

Other technological approaches can be used to study individuals in tight proximity. Salathe [83] used wireless sensor networks to study the transmission route of an infectious disease. They captured 762,868 interactions between individuals, determined the network between these individuals, and then subjected this information to a computer simulation of influenza spread. They determined that targeted immunization strategies are more effective than random immunization. Specifically, targeting individuals with high numbers of contacts, i.e., a strength-based vaccination strategy, had the greatest effect at curbing infection spread, but this effect was only present when vaccination coverage was sufficiently high.

2.3 Architecting More Powerful Systems: Wireless Sensor Networks, Macroprogramming, and Domain Specific Languages

Wireless sensor networks equipped to montior individuals and the areas in which they reside can provide significant insight into human behaviour and other health-related phenomena. Such systems, however, are difficult to use if researchers are forced to interact directly with their low-level coding constructs and underlying hardware [63]. These difficulties include limited battery life, low processing power, limited operating system support for higher-level operations, and the highly distributed nature of such networks. Not surprisingly, many authors have created frameworks purporting to allow easier, more intuitive, and faster control over these networks and the devices that comprise them.

A common approach to reducing the difficulty of interacting with these networks is to abstract them into a network-wide entity. These systems are termed *macroprogramming systems*, and the languages that power them *macroprogramming languages*. Macroprogramming can range from language extensions to existing frameworks to distributed abstractions such as database queries, Matlab-like matrices, and environmental entities such as “vehicle” or “doctor”.

2.3.1 Low-Level Macroprogramming and Language Extensions

The simplest macroprogramming systems are built by providing language extensions and low-level abstractions to the user. These systems are ideal when existing knowledge or infrastructure is to be exploited, but a simpler programming model is desired.

Hood [99] allows programmers to directly work with neighbourhoods of nodes in a wireless sensor network without the bookkeeping of neighbour lists, data caching, and direct messaging. Neighbourhoods are defined by the programmer using a definition of locality and any data to be shared. Nodes share values via a broadcast mechanism; receivers filter communications such that they store information only from neighbourhoods to which they belong and that they consider *valuable*. The exact definition of valuable is defined by the programmer and/or node’s logic. Hood is well suited to wireless sensor network applications in that it can cheaply broadcast information and expect other nodes to filter the information.

Hood defines several abstractions. *Filters* determine the nodes valuable enough to place in a node’s neighbour lists and which attributes of those nodes are to be cached for later use. A *mirror* is allocated for each valuable neighbour to give a local view of its state. *Reflections* in a mirror are cached versions of the neighbour’s attributes; *scribbles* are generally used to store derived values about that neighbour, such as distance estimate. These abstractions can be accessed via standard set of programming interfaces.

Kairos [37] allows a programmer to define the global behaviour of a wireless sensor network. The framework consists of a set of simple primitives – read/write variables, iterate through one-hop neighbours, and address arbitrary nodes – that abstract away details of distributed code generation, instantiation, data access, data management, and node interaction. These primitives allow for a surprisingly powerful range of operations to be performed. Algorithms implemented using these primitives at times resemble the pseudocode for, as the authors aptly put it, textbook algorithms.

Kairos acts as a preprocessor add-on for a target language and provides a runtime system for nodes in a network. This allows the programmer to use a platform-standard language, such as nesC [34], to configure nodes in a wireless sensor network. It also means that Kairos is language independent, and that its constructs could be used in almost any language capable of integrating a preprocessing system. Kairos' abstractions allow for concise expression of complex operations and have been tested in several example applications.

Kothari *et al.* [51] describe a system, Pleiades, to program a wireless sensor network via a central program. Pleiades is an extension of the C language that allows users to specify the global behaviour of their programs; when compiled, Pleiades programs are translated into nesC code appropriate for each device in the network. Pleiades' language extensions include concurrency primitives, node manipulation primitives, keywords to describe whether or not variables must be synchronized, and sensor abstractions. These extensions are coupled with program partitioning and program migration to appropriately compile and distribute their resultant nesC code. Deadlock detection, recovery, and distributed locking mechanisms are provided to facilitate concurrent actions.

EcoCast [96] is a macroprogramming system for wireless sensor networks. The system is conceptually simple, yet allows for powerful interaction with the network. The system works as an extension to the Python programming language. It extends Python's `map`, `reduce`, and `filter` functions to provide versions that work in a macroprogramming environment, allowing for simpler processing of distributed collections of nodes.

2.3.2 More Familiar Abstractions: Databases and Data Science Tools

Language extensions are useful to programmers familiar with wireless sensor networks, but do not drastically simplify the work they must undertake to perform certain tasks. When greater abstraction is desirable, familiar high-level abstractions are useful. The most familiar of these systems are those that emulate frameworks commonly used by researchers, i.e., databases, Matlab constructs, and data science tools.

TAG [59] allows users to interact with sensor networks using SQL-like queries. When queried, TAG requests data from the network and aggregates it as it passes through the network; this allows irrelevant, outdated, and redundant data to be removed during data transmission, which in turn reduces battery and processing difficulties faced by domain experts. TAG operates by distributing code that implements queries to all nodes. Once triggered, data collected through these queries is routed to a base station. Aggregation is performed as data flows to the base station, removing data that is not perceived to add value. Like many other macroprogramming languages, TAG reduces the amount of power and communication spent by the network when compared to centralized systems. It increases the level of abstraction that programmers can use to collect data, which can significantly simplify the data collection process. It also provides facilities to reduce the impact of network loss on aggregation results.

Data stored in wireless sensor networks can become stale, out of sync, inaccurate, or otherwise require processing to be useful to the domain expert. Nodes can determine when samples should be taken for an incoming query, which nodes are likely to have desired data, the order in which sampling should occur to most preserve power, and whether relaying a particular data sample is cost effective for the information it would provide. To exploit these facts and produce a familiar database-like abstraction for domain experts, Madden *et al.* developed TinyDB and Acquisitional Query Language [60].

Queries in TinyDB are much like those found in SQL. Data is selected from the sensors table, which represents data values at each node collected on an as-needed basis, or from materialization points, which are stable tables of collected sensor values stored on a single node. Data is generally collected over a sampling period on a duty-cycled basis. For instance, the following would acquire heat and humidity data from nodes over a 10 second period with one second between each sample:

```
SELECT heat, humidity
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

MacroLab [42] offers a vector programming abstraction like those presented by Matlab; this would appear to be a good choice for domain experts, as many scientists working with data are already familiar with this tool. MacroLab's statements are composed into microprograms that are run on individual nodes. The exact decomposition and distribution of these microprograms depends on the network topology, which in principle allows MacroLab programs to be automatically tailored to each deployment's application domain. MacroLab stores data in macrovectors. These represent sensor readings acquired over the whole network. Macrovector storage is dependent on the network topology: a highly centralized system might have all values stored on a single node or base station; a more distributed system might have a macrovector chunked and distributed across several nodes.

Program and macrovector decomposition is handled by the MacroLab compiler. Decomposition is based on cost information including network topology, power profile of nodes, radio hardware, and power restrictions; whichever decomposition scheme minimizes the final costs of the network is selected for use. This promotes code portability as developers do not need to tailor programs to new network types.

2.3.3 Region-Oriented and Region Monitoring Macroprogramming Systems

Many macroprogramming systems are region-oriented, i.e., they are tailored to identifying objects that exist in a space, tracking objects that move through a region, or facilitating the communication and grouping of network devices in a region at a high level. These abstractions are appropriate for researchers whose work is concerned with measuring population health in specific spaces, or for whom spatial factors play an important role.

Spatial programming is a programming model for sensor networks in which the space is key and resources are accessed with spatial references [10]. A *spatial reference* is an identifier encompassing the expected location of a node and a property it will provide. Borcea *et al.* implement this model using Smart Messages. Smart Messages are made of *bricks* that represent code and data sections to be migrated across nodes. In the Smart Messages framework, data is not migrated between nodes. Rather, bricks migrate to spaces and resources of interest and perform computations at the data source. Bricks are written in Java atop the Smart Messages framework, allowing programmers to interact with its abstractions via an extremely common programming language.

EnviroSuite [58] introduces a programming paradigm termed *environmentally immersive programming.* Environmentally immersive programming is not a language, but rather an object oriented-like paradigm that allows programmers to interact with environmental abstractions called elements. Elements are specified by sensory or geographic signature, a set of attributes, and a set of methods that may be performed when the element becomes available. Elements become available as their signature is detected. Straightforward examples are given in Table 2.1.

Abstract regions [98] was developed to simplify wireless sensor networks by abstracting away the details of communication, data sharing, and collective operations. The operators capture local communication. Local may take several meanings, including radio connectivity, spatial locality, or some other property of the nodes that may give them the label of local. All regions of a networks are able to identify neighbours and share data. Abstract regions allows easier communication without abstracting away accuracy and resource usage tradeoffs, which can be directly controlled by the programmer.

EnviroTrack [2] is designed to aid developers that must interact with and track components of the environment. This is accomplished through the use of an object-based middleware system. The system removes details of inter-object communication and provides automated object tracking functionality: when a node detects a user-defined entity, nearby nodes form a group around that entity and are given a context label, i.e., a construct that the programmer uses to interact with the entity.

To detect and manipulate an entity, three things are required. First, there must be a definition of what sensed values will represent the entity. Second, the programmer must define the state of an entity; states are generally aggregates from the group of sensors tracking an entity. Third, the programmer must specify

Table 2.1: EnviroSuite examples.

Element	**Signature**	**Attributes**	**Methods**
Vehicle	Metallic, noisy, high CO2 levels	Speed, location	Report speed, activate nearest red light camera
Forest	Within forest geofence, poor GPS and WiFi service	Light level, pollen levels	Report pollen levels, trigger pollen sampling unit
Space	No GPS service, no oxygen, high cosmic ray exposure	Tempera-ture, light level	Locate International Space Station, stream astronaut vitals to NASA, send SOS

the object that will interact with the sensed entity; this object will gain access to the state values that it produces and will be allowed to manipulate them. The state values produced contain freshness and critical mass thresholds, which tell the system how old sensor values can be before they must be disregarded and how many sensors are required for their aggregated state value to be valid.

Regiment [72] is designed to support spatiotemporal macroprograms, i.e., programs that need to work with significant spatial and temporal factors. These include programs that need to be aware of network topology, the geographical locations of nodes, and sensor data that varies given the time of day. Such scenarios typically require internode communication and estimating properties of portions of the network. This requires special constructs that may be absent in more abstract macroprogramming languages.

Regiment programmers view the network as a series of signals, which may originate from a single node or a region of nodes. Signals in Regiment represent a value at a given time. Functions can be applied to signals to produce further signals. Signals can be also be bundled into regions. Regions allow the programmer to work with an entire set of sensors and nodes that share some common property, e.g., all nodes reporting a high level of noise. The programmer is able to apply operations to all streams in the region without needing to handle communication mechanisms.

Motolla *et al.* developed a system called Logical Neighbourhoods [65, 66, 67]. Logical Neighbourhoods allows developers to define a node's neighbourhood based on node properties rather than their location. Neighbourhoods are defined using a language called SPIDEY. Each neighbourhood can be defined with a credit limit; this limits the energy and communication costs of neighbourhood definition, identification, and communication. The more credits that are given, the more ground the neighbourhood will cover and the more resources will be dedicated to achieving that coverage.

2.3.4 Logic- and Rule-Based Macroprogramming

Some logistical and monitoring problems are best represented through the use of logic and rules. For example, a health researcher looking for infection outbreak precursors may simply want to be alerted when the use of handwashing stations becomes abnormally high in an area. Several of such logic-based systems exist that can provide application-level updates, specify network-wide rules, or simply help researchers think about data in terms of interactions between entities that specify meaningful situations.

Semantic Streams [100] allows users to query networks for interpretations of data rather than querying the data itself. This is particularly useful for non-experts. Imagine, like the authors do, that a sensor system was created to monitor vehicles coming in and out of a parking garage. A non-expert would have difficulty working with raw magnetometer data and ascertaining vehicle types arising from the incoming data. The individual would instead like to know things like the number of cars and trucks entering the garage per day. Semantic Streams, through the combination of inference units created by other users, exists to provide this high-level information.

Semantic Streams operates on two concepts: event streams and inference units. *Event streams* represent timestamped incoming data and are fed into inference units. These can range from raw magnetometer data to inferred vehicle information. *Inference units* take event streams and convert them into more useful information. For example, an inference unit might take `magnetometer` and `height` data reported by event streams and use them to infer vehicle size. This would produce another event stream, which could be used by another inference unit coupled with a `weight` event stream to produce a vehicle estimation. Streams and inference units can be coupled with constraints, e.g., a minimum degree of certainty, to limit their results and output. Streams and inference units are defined using logical predicates that are similar in syntax to Prolog.

FACTS [95] provides a rule-based approach to event processing to reduce the difficulty programmers face when programming event-based systems, e.g., notifying a nurse when a ward becomes devoid of other nurses for a significant period of time. FACTS is based on rule, fact, and function abstractions. Rules express algorithms and reactions to events. Facts are used to represent data. Functions allow rules to interact with software outside of FACTS' middleware. FACTS' rule engine uses forward chaining – as opposed to backward chaining found in many rule-based systems – to provide event semantics. These abstractions are local to each node, but are used to provide information, such as new facts, to other nodes, the network, or node groups.

Frank *et al.* [30, 31] describe a role specification language to configure a sensor network that ensures user specifications are met. Role specifications include a rule and identifier. The rule gives the conditions that must be fulfilled on a sensor node and its neighbourhood for a role to be enacted in that environment. Roles are defined using logical predicates. Special predicates, i.e., count and retrieve, allow the programmer to interact with other nodes in the network.

Chu *et al.* [18] describe a declarative sensor network platform. Their platform uses a variant of the declarative Datalog language, Snlog, that can interface with external code to perform non-declarative actions. Snlog consists of predicates, tuples, facts, and rules. The authors provide a representative yet simple example:

```
% rule
temperatureLog(Time, TemperatureVal) :-
  thermometer(TemperatureVal),
  TemperatureVal < 15,
  timestamp(Time).


% facts
thermometer(24).
timestamp(day1).
```

`temperatureLog(Time, TemperatureVal)`, `thermometer(TemperatureVal)`, and `timestamp(Time)` represent predicates. When Time and TemperatureVal are given values, these predicates form tuples. Tuples that are statically given or provided by the system, such as thermometer(24), represent facts. Rules are defined by combining predicates in conjuntive normal form. Predicates and rules interact with the sensor platform and physical world by invoking native code using the `builtin` predicate, e.g., `bulitin(temperature, 'TemperatureSensorReader.c')`. Tuples that need to reference other nodes specify this intent by prefixing variables with the @ symbol, e.g., @Node7. Snlog can reference external code: their language can be used to think about how to solve problems and acquire data while leaving other languages to implement the "plumbing" needed to acquire information for fact resolution.

CRIMESPOT [22] is a declarative rule-based language that operates atop specialized middleware. It contains both node- and network-level programming facilities, allowing developers to specify how nodes should respond to rules and which rules should govern different parts of the network. CRIMESPOT also provides temporal facilities for working with data. Unlike many rule-based languages, CRIMESPOT uses forward chaining rather than backward chaining to evaluate rules: when the fact base changes, rules dependent on these facts are re-evaluated.

Bischoff *et al.* [9] describe the macroprogramming framework named RuleCaster. RuleCaster uses the language RCAL to specify state transitions for a sensor network. RuleCaster's compiler takes programs written in RCAL and determines where to place state manipulation and storage modules. RCAL applications are based on ruleblocks. Ruleblocks are defined using Prolog-like rules. Each rule defines the space in which it operates, the time interval at which it is evaluated or the state transition that must occur to trigger evaluation, and further state transitions that occur when the ruleblock is affected by the transitions on which it depends.

Applications are not necessarily interested in monitoring sensor values to detect context changes. Rather, applications are interested in high level information that will change the way they operate, such as the beginning of participant physical activity, sudden falls in temperature, or the beginning of a forest fire. Context monitoring queries (CMQ) [45, 46] were developed to deliver notifications of these changes to applications. A CMQ is given by specifying the context to monitor, the boolean transition on which it is to fire an alarm, and the duration for which the query should remain active:

```
CONTEXT <context element> (AND <context element>)*
ALARM <type>
DURATION <duration>
```

The CONTEXT clause specifies the condition to monitor in conjunctive normal form. The ALARM clause specifies if a program should be notified when the condition goes from True to False or from False to True. The DURATION clause specifies how long the query should remain active.

Applications register queries with a CMQ Processor provided by SeeMon [45, 46], which implements and manages the queries. The system's Sensor Manager then closely examines each component to find an Essential Sensor Set, which allows SeeMon to reduce the amount of power and computing overhead required to monitor context changes and allows SeeMon to be extremely scalable. The exact computation of an Essential Sensor Set is quite involved; interested readers may consult the authors' solution [46].

Pathak *et al.* [75, 76] developed a programming model called ATaG. ATaG is a data-driven macroprogramming language. Programmers define high-level operations in a query format; the compiler then assembles these queries into lower-level units of code, distributes them to appropriate nodes, and begins execution of the programmer's query. ATaG manages control, coordination, and state maintenance of the wireless sensor network on which it runs. ATaG is a declarative-imperative language. Its core abstractions are the *abstract task*, which process data, and the *abstract data item*, which are processed by tasks. Abstract tasks are connected via *abstract channels*, which connect tasks requiring input to tasks that output the required type of data. Programmers express interactions between tasks not with channels, but with the abstraction of a shared data pool. Tasks output data to this pool and are register to be notified when data of interest is deposited by other tasks.

Abstract tasks may encompass any type of logic, including localized data sampling tasks, data compression, and network-wide data aggregation. Tasks are attached to firing rules that dictate their operation. These include duty cycles and rules based on data availability. Tasks are deployed to sensor nodes automatically based on a set of criteria specified by the programmer.

At a high level, ATaG programs are provided to the compiler by specifying three items. First, the abstract tasks, data items, and channels must be specified. Second, the imperative logic of each abstract task must be provided. Third, a description of the target network must be provided. The description must provide enough information such that any task specifications, e.g., required area per task instance, can be fulfilled. The output of the compilation procedure [75] is a program that uses the Logical Neighbourhoods API [66].

2.3.5 Middleware, Agents, and Injectable Code for Programmers

Many sensor network frameworks for collecting data on individuals, populations, and areas aim not to provide a specific programming language, but rather a better development environment, better tools for reconfiguring deployed networks, and better deployment management. These tools are relevant to this thesis as they provide alternative paradigms and methodologies for developing data collection architectures. For example, instead of pre-setting devices with a master architecture and configuration, Agilla [28] consists of migratory agents that move to the area of the network that need their attention; instead of building a data collection framework with support for a single language, SwissQM [69] acts as a virtual machine through which any language can interact with the underlying sensor system. Several of these systems could provide direct benefits to the work of this thesis if someday integrated with it.

Li *et al.* [53] developed a middleware system named DSWare. DSWare provides a database-like abstraction to applications and allows events to be detected using a combination of confidence functions and sensed data. Confidence is based on the reliability of collected data and the correlation between data and specific events. DSWare exists between the application and network layers of sensor networks, helping applications avoid reimplementation of common data services. Data describing occurrences of an activity can be mapped to a location and cached, speeding queries for this data. Group management services are provided to help nodes cooperate and to detect erroneous or strange data being reported by individual devices. Energy-aware and real-time scheduling are provided. Event detection and data subscription services are provided by DSWare, allowing apps to get the data they need in an efficient and timely manner without a great deal of effort.

TeenyLIME [21] is a tuple space middleware for collecting and processing data within a sensor network. TeenyLIME is based off the LIME framework, which in turn is based off of Linda, a tuple space manipulation language. LIME, and hence TeenyLIME, manipulates a distributed tuple space whereas Linda uses only a single, shared memory tuple space. TeenyLIME differs from LIME in that each node can only see the tuple space of one-hop neighbours; the tuple space visible to each node, therefore, is almost certain to be different from others in the network. In TeenyLIME, developers can insert, read, and remove tuples from the tuple space. They can also react to tuples as they arrive. These operations are blended into nesC code, which can be used to empower existing applications are produce new ones.

Ocean *et al.* [73] describe SN BENCH, a system for aggregating disjoint sensor networks (SNs) into a larger, virtual SN. This increases the level of abstraction for network programmers when hooking into existing networks. SN BENCH further simplifies this situation through SNAFU, a functional network programming language. SNAFU compiles to Sensorium Typed Execution Plans (STEPS) – a task-oriented language used to assign work to individual SNs – and is used to implement the developer's needs. SN BENCH also provides facilities to monitor SN resources, schedule new STEPS as required, and unifying interfaces to similar hardware components.

SNAFU is a strongly typed functional language. It is used to specify dependencies between sensors, computational resources, and states within an application. SNAFU is a macroprogramming language: programs are written for the SN with SN BENCH handling dissemination and resource allocation for the resulting programs. Unlike many functional languages, SNAFU does not support recursion; it instead provides iterative execution through the trigger construct, which are executed when a given predicate becomes true. There are two major classes of triggers: terminating triggers specify a predicate that, once true, causes the trigger to execute and then never execute again; persistent triggers constantly evaluate their predicate and, when the appropriate predicate transition is detected, execute. SNAFU compiles to a directed acyclic task graph. Each node in the graph is a value, sensor, or task to be performed; edges specify dependencies and communication. As each program is represented as a DAG, nodes can be easily computed and their complexity will be known as runtime. These graphs are stored in STEPS, which are themselves encoded using XML documents. This allows STEPS to be interpreted by individual SNs instead of imposing a top-down command language. The compilation and distribution of STEPS is well outside the scope of this thesis and are therefore not described here.

SwissQM is an architecture for sensor networks [69]. It is based on a virtual machine that runs on bytecode rather than queries. Much like the Java Virtual Machine, this allows users to specify whatever language they want to interact with the underlying sensor system; this allows many languages to be implemented on the same platform and run at the same time. These systems, however, must be able to translate their language constructs to SwissQM bytecode.

Boulis *et al.* [11] assert that it is desirable to be able to dynamically configure a wireless sensor network without having to distribute a pre-formed executable image. They approach this problem through the use of SensorWare, which allows new services to be injected into the network, distributed to nodes, and executed where appropriate in conjunction with other deployed services. SensorWare uses TCL as its scripting language to implement this functionality. It provides four primitives, query, act, interest, and dispose, to interact with virtual devices. New devices can be created at runtime, which changes the runtime environment for future scripts by allowing access to new abstractions and devices.

COSMOS [6] is an architecture for wireless sensor networks composed of an operating system and programming language (mPL). System behaviour is defined as dataflow processing: programmers provide functional components (FCs), which represent elementary units of execution, that are combined together to perform larger actions. FCs produce and/or respond to data asynchronously and are non-blocking; when data is available, they are scheduled for execution, which reduces the system's memory footprint. FCs are highly reusable and can be plugged into any system that requires them.

FCs are defined in the mPL language. They are composed of the FC declaration and an implementation provided in a subset of the C language. When a macroprogrammer uses an FC, only the declaration is important, which allows a hybrid design for new programs: expert programmers can write necessary logic in C, whereas application programmers can simply use the higher abstractions. When mPL is used to design FCs, several expressions can be used, including enumerations, instantiation of FCs and devices, capability constraints, FC and device input/output signatures, and contracts.

Abdelzaher *et al.* [14] provides a description of LiteOS. LiteOS tries to simplify wireless sensor network programming by providing Unix, thread, and C abstractions. Sensor network nodes are mapped to file directories, allowing them to be used in a fashion familiar to most programmers. Once connected to a base station and mapped, programmers can manipulate nodes via a command line shell. For instance, executables can be run, processes can be viewed, and node contents can be copied in an interactive fashion. While perhaps not suitable for domain experts, the system provides an interesting abstraction for working with sensor networks.

Fok *et al.* [28] developed a middleware system named Agilla. Agilla allows agents, i.e., units of code that can perform work, to be injected into a WSN. These agents are autonomous and encapsulate a task. They are defined in a high-level language and communicate via Linda-like tuple spaces. Agilla networks begin with no preinstalled software. Instead, agents migrate across nodes, populating nodes as required and dying if they are no longer needed. This allows Agilla networks to be reprogrammed.

Each node maintains a tuple list, neighbour list, and a list of all agents running on it. The agents running on each node can access each other's tuple space, but do not carry these with them; if an agent requires data from its previous node, it requests it. In addition to processing tuples and the data they represent, agents can react to tuples as they enter the tuple space, allowing for efficient event processing over blocking or polling.

Agilla's architecture is composed of an Agent Manager, Context Manager, Tuple Space Manager, and Agilla Engine. The Agent Manager maintains each agent's context, i.e., maintaining its memory space and determining when it is ready to run. The Context Manager determines the location of the agent and its neighbours. The Instruction Manager allocates instruction memory for agents. The Tuple Space manager implements and manages tuples, reactions, and tuple operations. The Agilla Engine controls concurrent agent execution as well as their arrivals and departures.

2.4 Literature Surveys and Design Guidelines

The architecture and language developed in this thesis do not exist in a vacuum and are not the first of their kind. Taxonomies and classification schemes exist to group existing architectures, and several design guidelines exist for building configuration languages. As the remainder of this thesis focusses on the design of a data collection system and its accompanying language, the related work analyzed in this section serves to set a high-level context for the work that was performed. They are further referenced in chapter 5 following the description of the system to give the reader a sense of where it fits amongst the existing literature.

2.4.1 Sensing Systems and Wireless Sensor Networks

The majority of deployed wireless sensor network applications use low-level systems programming despite much discussion regarding higher level approaches. Mottola *et al.* [68] created a literature survey to summarize the high-level systems that have been proposed to help reduce the difficulties of low-level programming. They list several common traits of WSN applications and the major categories they fall into:

Goal. The goals of the WSN application: sense-only, sense-and-react.

Interaction Pattern. How nodes in the application interact: one-to-many, many-to-many, many-to-one.

Mobility. How nodes and data sinks move in an deployment: all entities static, mobile nodes, mobile sinks.

Space. The type of distributed data processing in space: global, regional.

Time. The type of distributed data processing in time: periodic, event-triggered.

A taxonomy of languages that power these systems is also given. The taxonomy is rather large and most conveniently described using the table found in their work; a brief overview, however, is given in Table 2.2 of major high-level categories. An architectural taxonomy is also provided in Table 2.3.

Domain experts often have little experience with sensor networks and become frustrated by the languages that drive them. Bai *et al.* [7] addressed this problem through two contributions: a taxonomy of design features required for sensor network deployments, and an example system, named WASP, for novice users of sensor networks.

Table 2.2: Summary of language taxonomy [68]

Category	**Description**	**Subclasses**
Communication Scope	The nodes that communicate to accomplish the goal of an application.	Physical neighbourhood, system wide, multihop group
Communication Addressing	How nodes are identified.	Physical, logical (i.e., application-level properties)
Communication Awareness	How aware the programmer must be of communication facilities. Specifically, must the programmer communicate explicitly between nodes, or is this provided implicitly through higher-level constructs?	Explicit, implicit
Computation Scope	The sphere of influence that a particular instruction can have on the network.	Local, group, global
Data Access Model	The abstraction by which data is accessed.	Database, data sharing via variables or tuples, mobile code (e.g., agents), message passing
Programming paradigm	The paradigm used to write programs in a WSN language.	Imperative, Declarative, Hybrid.

Table 2.3: Summary of architecture taxonomy [68]

Category	**Description**	**Subclasses**
Program Support	"The extent to which a given programming abstraction provides support to the programmer."	Holistic, Building Blocks
Layer Focus	The architectural layers that are the main focus of the programming approach.	Application-level, vertical solutions (i.e., all layers).
Low-Level Configuration	How the approach provides access to low-level layers of the WSN.	Fixed at compile-time, dedicated interfaces
Execution environment	The platforms that are supported. Some systems have only been reported to be used via simulation.	Real Hardware, Simulation

Through studying 23 sensor network applications, Bai was able to extract 19 application properties – only eight of which directly affect language design – and developed an archetype taxonomy. These properties are as follows:

Mobility. Mobile nodes need to understand where they are and how they are moving.

Sampling Initiation. Nodes may have data collection that is initiated periodically, by events, or both.

Data Transmission Initiation. Like sampling, applications must have the ability to specify how and when they transmit data to a base station.

Actuation. Nodes that can trigger hardware or environmental actions need to be able to initiate those actions and specify the conditions under which they will be invoked.

Interactivity. Systems whose nodes are to respond to commands and queries mid-deployment must have support for those commands, queries, and reactions.

Data Interpretation. Systems that support in-network data interpretation require the ability to specify data processing procedures.

Data Aggregation. Like systems that support data interpretation, those that support aggregation require the ability to specify data aggregation procedures.

Homogeneity. Networks made of multiple node types (i.e., heterogeneous networks) require the ability to distinguish between node types.

Bai simplifies the combinations of these facets to form seven archetypes that cover the majority of sensing applications. The archetype table they present is reproduced in a slightly altered form in Table 2.4 to improve readability. ANY indicates that a cell may take any value.

These archetypes represent broad categories of sensor networking applications based on functional properties. Their properties shape the syntax and style a language should take if it is to be suitable for these types

Table 2.4: Modified table of archetypes [7]

Archetype	**1**	**2**	**3**	**4**	**5**	**6**	**7**
# Examples	7	6	4	3	1	1	1
Mobility	stationary	stationary	mobile	mobile	stationary	stationary	mobile
Sampling	periodic	ANY	periodic	periodic	hybrid	event-based	periodic
Transmission	periodic	event-based	ANY	ANY	hybrid	hybrid	event-based
Actuation	N	N	ANY	ANY	N	Y	N
Interactive	N	ANY	N	Y	Y	Y	N
Interpretation	ANY	Y	ANY	ANY	Y	N	Y
Aggregation	ANY	ANY	ANY	N	Y	Y	Y
Homogeneous	Y	Y	Y	N	Y	Y	N

of problems. Archetypes also have benefits beyond language design: they can be used to provide problem solution templates to new users, and can instruct them in the language to use for their problem. As Bai shows, simply having a template can improve novice task solving success rates with a new language from 0% to 8.3%.

Bai's language, WASP, represents a system in archetype 1. WASP lets the user view the network as a set of distribted data arrays. Each array represents a node-level variable and the stream from which data is derived. New data is placed into the array at the top, pushing older data down; sensors can simply index into the array to find the data they need. WASP further allows users to specify data processing operations at the node and network level for greater expressiveness.

A WASP program is broken into local and network segments, which provide support for local node and network wide behaviour, respectively. Local-node syntax follows a `SAMPLE sensor EVERY time time_unit INTO location` pattern, with support for arithmetic expressions, function calls, and scheduled versions of these constructs. Network syntax follows that of TinyDB [60] and provides a `collect-where-group by-having-delay` clause-based syntax.

2.4.2 Domain-Specific Languages: Experience-Driven Design

While the authors above focussed specifically on the design of systems and languages in the context of wireless sensor networks, other authors have expanded on encompassing idea of domain-specific languages. A domain specific language (DSL) is a "programming language tailored specifically to an application domain: rather than being for a general purpose, it captures precisely the domain's semantics" [87]. For example, a domain specific language could be created to specify how health information should be retrieved from a hospital-wide monitoring network. In-the-field experience has been used to generate several of these reviews, some of which are described below.

Spinellis *et al.* [87] describe design patterns for DSLs. They begin with a description of DSLs. According to Spinellis, a domain specific language is "a programming language tailored specifically to an application domain: rather than being for a general purpose, it captures precisely the domain's semantics." DSLs allow an application's logic to be described concisely, expressively, and appropriately for the target domain at the expense of generality. They involve domain experts, which allows these individuals to be actively and routinely involved in the software development lifecycle. They can be, however, difficult to integrate into a group's larger software processes and require that other individuals be trained in their usage.

There are several design patterns noted in Spinellis' work. As the work of this thesis encompasses a domain specific language that uses some of these patterns, they are listed here:

Piggyback. The DSL uses the existing language's capabilities to implement its own features. The DSL effectively acts as a frontend to the fuller language being exploited.

Pipeline. The DSL is part of a larger system whose subcomponents are best described with a specialized syntax.

Lexical processing. The DSL offers shorthand notation(s) for expressing logic or entities that are later replaced by lexical substition, e.g., replacement of the shorthand text using a regex or the C preprocessor.

Language Extension. The DSL is used to add new features to an existing language.

Language Specialization. Features are stripped from a base language to form a DSL, allowing it to be used for a specialized purpose.

Source-to-Source transformation. The DSL is used to translate code artifacts of one type to code artifacts of another.

Data Structure Representation. The DSL allows complex or domain-specific data structures to be declaratively specified in a user-friendly fashion.

System Front-End. The DSL is used to configure and adapt a system for a particular use case.

Wile *et al.* [102] discussed their experience constructing DSLs for real-world communities. They describe several lessons that have been learned from these experiences, paraphrased here:

1. Adopt notations and jargon used by domain experts rather than inventing a new one.
2. Understand the roles and scope of the people who will use your language, and understand the existing solution(s) to the problem to be addressed before introducing a DSL.
3. You are likely not designing a programming language, so design only what is necessary.
4. Strive to solve 80% of the domain expert's problem.
5. Leverage the infrastructure you are providing to reduce language complexity.
6. Strive for improvements in domain expert productivity.
7. Many specification techniques may be necessary to accommodate domain knowledge and techniques.
8. Work closely with domain experts to produce infrastructure that the system will translate to.
9. Understand the background of those to be affected by the DSL – they might not be able to understand a DSL or maintain it.
10. Make sure that the DSL and associated technologies are consistent with your client's business model.
11. Understand your development team's weaknesses and plan to deal with them.

12. Don't expect that domain experts will know what the system can, should, or cannot ever do for them, and don't expect them to forgive design mistakes.

Mernik *et al.* [62] state that DSLs make problem solutions easier to write and express in a specific problem domain while reducing generality. DSLs have existed for a considerable length of time with several languages hailing back to the late 1950's.

Mernik states that DSLs are developed for several reasons. They may use domain-specific notations that are appropriate for users and problems in the domain. They can offer domain-specific constructs and abstractions that are not easy to write in general programming languages (GPLs). They can offer analysis, verification, optimization, and parallelization opportunities through abstractions in the underlying GPL. Their development and uptake, however, can be limited given the costs associated with their creation, maintenance, and upkeep, and the risks associated with taking up obscure or poorly supported DSLs.

DSLs can but do not necessarily need to be executable. Some DSLs, such as HTML, have "well-defined execution semantics" [62]. Others are used to generate applications. Some, like BNF, can be used to specify a problem, but can also be used to instruct a parser generator. Some DSLs are simply not meant to be executable at all [102].

Mernik states that DSL development tends to follow a lifecycle containing decision, analysis, design, implementation, and deployment phases. They proceed to elaborate on these phases, beginning with a list of several decision patterns in which DSLs use is justified. These are listed to help the reader determine if a DSL is appropriate for their circumstances. These include the following:

Notation. Domain specific notations must be represented in a program.

AVOPT. Analysis, verification, optimization, parallelization, and transformation are required but infeasible with existing GPL constructs.

Task Automation. Tedious tasks following similar patterns frequently arise, but the tedium could be removed or simplified through automation.

Product Line. Related software would benefit from a shared specification tool.

Data Structure Representation. Data-driven code requires data structures to be initialized, but this is difficult to do and maintain in a GPL.

Traversal. Easy traversal over data structures is required.

Interaction. Designers want to make complex or repetitive input quick and easy to specify.

GUI Construction. A GUI must be built, and it is difficult or time consuming to do so in a GPL.

Abelson [3] describes the uses of Lisp and, in particular, its flexibility. Lisp is both one of the oldest computer languages still in use and has a syntax flexible enough to produce within-language DSLs. Its power lies in its ability to express robust designs for computer systems, i.e., designs in which small changes in system requirements can be made by only small changes in the design. The language also allows constructs to be expressed in a representation natural to the problem domain – and can even temporarily or permanently transfer control to external programs – using metalinguistic abstraction. Lisp programmers can construct interpreters for new languages within Lisp, and then use those languages within Lisp itself. It could be argued, however, that Lisp syntax is not easily parsed by the non-expert despite its ability to express a variety of special-purpose languages within its own syntax.

2.5 iEpi: Previous Work

The language and data collection system developed in this thesis are part of a larger research project named iEpi, which has been in development for several years. It was created to replace and extend a precursor system named Flunet. Flunet [40] was a wireless sensor network intended to capture human contact data throughout flu season. Thirty-six participants were asked to carry a Flunet-enabled MicaZ mote for a period of three months. Participants were further asked to fill out health-related surveys each week. The identified contact patterns of individuals are stated to be similar to that of Reality Mining [23]. This result is interesting in that two different technologies converged on similar conclusions.

The creation of iEpi was sparked by Flunet for several reasons. First, Flunet's creators found that motes were easily damaged. This was because many motes are not designed for human use. Smartphone-based technologies, however, are designed with human-induced abuse in mind. Creating a Flunet-like system on a smartphone platform would increase system robustness. Second, motes are easily forgotten, ignored, allowed to discharge, or otherwise rendered temporarily useless by users. Smartphones are less prone to these problems and, when they do occur, are typically resolved quickly because users generally take care of their smartphones, ensuring that they are charged, protected from damage, and normally on their owner's person. Third, motes proved difficult to program. Flunet's creators once remarked how their debugging process involved three LEDs and no logging capabilities.

Smartphone programming environments are significantly more friendly in this regard. Smartphone environments are also coupled with a significantly larger support community, which eases the development process. Finally, smartphone-based technologies are significantly easier to distribute to human populations. Delivering motes to participants works well for small-scale studies, but does not scale well. For example, it would be difficult to maintain a study of 1000 participants with mote-based technologies, but a smartphone-based technology would simply need to be posted to the relevant platform's app store.

iEpi was further motivated by difficulties in understanding the spread of infectious disease. The spread of these diseases can be indirectly studied by using aggregate, mathematical, and agent-based computational models. Models, however well constructed, can unfortunately become inaccurate without fine-grained data. iEpi attempted to provide and use fine-grained behavioural data to improve the quality of influenza-like disease modelling. The system's earliest incarnation [39] was used to collect data over five weeks from 39 participants. While significant battery problems were encountered, e.g., data transfer bugs that would periodically reduce phone battery life to only a few hours, 45 million data records were collected. Similar work has since been performed [38] with data analysis still ongoing.

These early studies utilized precursors to iEpi's current architecture (chapter 4) and language (chapter 5) capabilities. The logger, i.e., iEpi's smartphone data collection component whose architecture is the subject of chapter 4, possessed a simple *stream-task-data logger-upload-database* pipeline. *Streams* generated data. *Tasks* encompassed streams and controlled the time at which they generated data. When data was collected, the *data logger* was invoked to store data, which was later retrieved for upload and storage in an off-device database. The data logger was also capable of invoking external applications and uploading their data. At this point in its development, iEpi used an extremely restricted version of Seddacco (originally named *iEpian* [38]) for its configuration. Each type of task had to be defined in its own file, which were then parsed with file-specific regular expressions to extract tasks to be instantiated.

Seeing iEpi's potential for uses outside of the epidemiological realm, collaborations with outside departments were performed. In particular, a collaboration was undertaken with the University of Saskatchewan's Geography Department in order to better understand human mobility and indoor localization. iEpi, ArcGIS, SaskEPS, and Walkable CentreLINE were employed to record and map the activities, movement, and interpersonal contacts of individuals [77, 78]. The combination of these tools allowed individuals' movements to be snapped to paths across the University of Saskatchewan's campus much like GPS tools do with roads and streets. The logger's role in this experiment was to collect fine-grained data in order to make reliable indoor positioning estimates. Since the devices on which the logger ran needed to operate for an entire academic day, data collection and upload behaviour needed to be carefully tuned. The logger's flexible duty cycle and data collection burst mechanisms were heavily employed to accomplish this. In the case of upload and update capabilities, which are potentially power heavy operations, power assessment modules (referred to as *conditions* in chapter 4) were also used. As data collection occurred every two minutes for thirty seconds, i.e., the phone was in a high power state for 25% of the day, phone batteries did not always last as long as was desirable. They did, however, last long enough to produce interesting traces of individuals over a four week period.

During the collaboration with the Geography Department, phone behaviour was specified using a version of Seddacco (chapter 5) that employed TXL [85] to implement a rudimentary interpreter. Seddacco programs were translated using TXL into an XML intermediary, which was then parsed and interpreted by iEpi. While this approach was limited and difficult to extend, it provided experience working with compiler-like systems, and was more reliable and flexible than the regular expressions used in previous experiments [38, 39].

Chapter 3

iEpi

This thesis concerns the architecture [49] and configuration language [50] of a data collection system named iEpi [38, 39]. In the following chapters, these components are discussed in detail. To give the reader an understanding of the system as a whole so that these components may be understood in context, this chapter provides a brief overview of the larger system and the capabilities it provides.

3.1 Overview

iEpi (Figure 3.1) is an end-to-end system for collecting and analyzing contextual microdata using a chain of smartphone, desktop, and server technologies. Data is collected via several smartphone components. This set always includes the "logger" (chapter 4), which acts as the primary data collection and processing engine of sensor and contextual information. It is complemented by a survey tool and, to a lesser extent, meal, weight, and activity logging software. Collected data is encrypted, compressed, and uploaded to a server. There, it is processed by the Autoparser, which decrypts the data and inserts it into an appropriate database. Server side tools are then used to process, interpret, or otherwise utilize the data. These include simulations and models developed in tools such as AnyLogic [20], ArcGIS [27], and R [80]; indoor positioning technologies; battery analysis scripts; and a variety of custom-purpose database scripts.

3.1.1 The Logger

The logger is a smartphone app that collects data from individuals. It uses a highly parallelized, pipeline-based architecture [49]. Each pipeline is composed of tasks, which are data processing units. Each task may generate, filter, transform, or otherwise manipulate data before giving it to the next task in the pipeline. Each pipeline is scheduled according to the user's needs and limited to desired operating contexts, if any. The logger is configured using the domain specific language Seddacco [50], which automatically connects tasks, schedules, and context limitations using a concise, extensible, English-like syntax.

The logger frequently interacts with external tools and sensors. The most commonly used tool is the SurveyApp [38]. The SurveyApp displays surveys specified in XML files. The logger has also interacted with the photo logging software PhotoFoodDiary, Bluetooth-powered weight scales, and even EEG devices

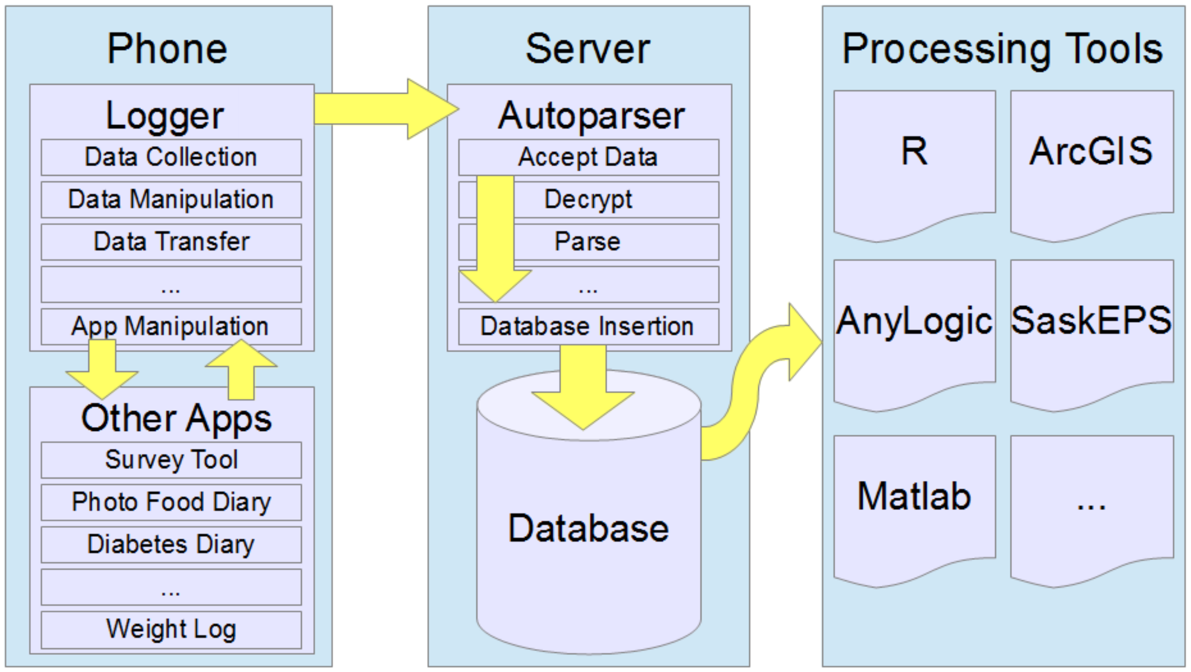

Figure 3.1: The iEpi System Overview

[38, 39]. Its user interface is often modified and extended to provide study-specific functionality, e.g., input forms for blood glucose levels and insulin doses.

3.1.2 The Autoparser and Backend Servers

The Autoparser is a tool that takes iEpi-related data, processes it, and stores it in a database. The Autoparser is primarily used to process logger- and SurveyApp-generated files, but can be extended to accept other data formats. The Autoparser's main abstraction is the *data connection*. Data connections map named fields in input data to columns in a target database. They are configured using XML, allow the Autoparser to automatically generate database tables, and allow users to parse data without needing to reprogram an existing tool – provided that no non-standard, pre-database processing operations are required.

The Autoparser runs on a DMZ server and sends extracted data to a dedicated database server. A DMZ server acts as a bridge between the wider, insecure internet and a secure internal network; in the context of iEpi, the DMZ server hosts and accepts uploaded files, facilitates file decryption and parsing, and transfers data to a secure database server disconnected from the internet. The user accesses the secure database server from the local network on which it runs.

3.1.3 Data Analysis

Data that has been collected and placed on the secure database server is processed by a variety of tools. These tools are beyond the scope of this thesis, but include simulation models written in AnyLogic, Rubybased battery analysis scripts, state and activity classification algorithms, trilateration and localization tools written in C# and ArcGIS, and a variety of scripts written for statistical software packages such as Matlab and R.

3.2 System Capabilities

Given references to the size, generality, and extensibility of iEpi throughout this thesis, the reader might rightfully wonder what system capabilities currently exists and for what the platform can be used. While this seems like a straightforward exercise, difficulties arise from the sheer extensibility of iEpi. For example, the logger was built to collect survey and embedded sensor data (e.g., GPS), yet its design makes it relatively straightforward to directly integrate EEG (i.e., brainwave reading) devices and augmented reality camera views to its core data manipulation arsenal. To give a sense of what iEpi is capable of, a summary is provided below.

3.2.1 Recording Capabilities

iEpi is capable of recording data from the following types of sensors and on-board devices:

Proximity

Knowing the proximity of a person to another person or a specifically tagged location has ramifications for the study of public health, epidemiology, sociology and social psychology. iEpi primarily infers proximity by using the strength of Bluetooth radio signals as a proxy for distance to nearby devices.[1]

Bluetooth allows the device to detect nearby devices, determine their type (e.g., computer, smartphone or radio-controlled toy), and estimate their degree of proximity. This can be used as a proxy for human proximity, as smartphones are often on the person of or proximate to their owners. Phones can only be detected if they are in discoverable mode, which is enabled by iEpi configurations employing Bluetooth. Modern phones are by default non-discoverable to save power, so users not using Bluetooth monitoring via iEpi must specifically enable discoverable mode, providing implicit consent for monitoring. A weak estimate is that 7% of cellphone owners and 5% of smartphone owners leave their phones in this mode, allowing iEpi to detect at least some non-participants while conducting a study.

Anonymization

Bluetooth devices are identified by a unique media access control (MAC) identifier. This allows researchers to determine repeated contacts with the same device, which is highly desirable for understanding contact patterns for a number of human behaviour studies. To ensure that phones can be identified within the study

[1]Ignoring environmental confounds, a weak signal suggests that a device is far away; a strong signal suggests that a device is nearby.

but not after the study, all Bluetooth MAC addresses are hashed to a new ID. For participants, these IDs are tied to participant names and may be stored in an encrypted file on the phone for use in surveys about social networks. Non-participants who have their phones in discoverable mode will receive the same hashed ID throughout the study, but it will be impossible to determine the MAC address of their phone or their identity. After a study is complete, the key used to generate the hashed IDs is deleted. This de-identification step allows consistent labeling of phones yet provides no method of directly identifying the participants generating the labelled data.

Location

A person's location in space is a fundamental variable of interest for many human behavioural phenomena and is particularly important for health research. Several methods from which location estimates can be derived exist, and not all methods are appropriate in all environments. iEpi therefore collects location data in a variety of ways:

- **GPS.** The acquisition of GPS (or assisted GPS – AGPS – if a network connection is available) allows iEpi to detect the device's speed, heading, and location. The location can provide as fine as five meter resolution outdoors (as measured by standard deviations of signal estimates), or in low density indoor settings.

- **Celltower Data.** Proximity to celltowers can give a coarse representation of the location of a participant by reporting the celltowers to which that participant is currently connected. This method is clearly inferior to GPS in terms of spatial resolution, but may occasionally be collected as a comparison point for studies based on call logs, which provide larger sample populations but coarser spatial resolutions.

- **WiFi.** WiFi data providing the signal strength and identity of detected routers allows the device to identify nearby WiFi access points. The physical location of access points can be derived from for-profit external services, databases maintained by facilities, such as the one maintained at the University of Saskatchewan, or estimated from coincident GPS measurements. When several WiFi access points are detected, iEpi can perform post-hoc trilateration (i.e., determine location). Trilateration accuracy varies with WiFi router density. At high router density, consistent with the University of Saskatchewan, localization within six meters is possible, particularly if building layouts are available to limit the number of possible locations.

- **Bluetooth.** Just as signal strength from mobile Bluetooth devices can be used as a proxy for personal proximity, signal strength from stationary Bluetooth devices can be used to infer location. This modality will become particularly important after the full commercial release of the iBeacon [43], which is an inexpensive Bluetooth device for tagging locations of interest.

Activity and Motion

While the first two measurements deal with where a person is and who they are with, activity and motion detection deal with understanding what a person is doing at a location or with someone else. Despite many activities being well outside the measurement capabilities of a smartphone, many important behaviours and health behaviours – such as sitting/standing, sedentary/active, walking/running or even if the phone is being carried by the participant – can be inferred from motion sensors on the device. iEpi attempts to incorporate as many of these sensors as possible into its repertoire, and leaves architectural support for more to be integrated as needed:

- **Accelerometer.** The accelerometer measures acceleration on three axes. As gravity provides a constant downward acceleration, these sensors can also infer the orientation of the phone with respect to the surface of the Earth.

- **Gyroscope.** The gyroscope measures the rotational speed of the phone. This information is typically used in conjunction with the accelerometer to determine the orientation of the phone.

- **Magnetometer.** The magnetometer measures the local magnetic field. This is typically used in conjunction with an estimate of the phone's orientation to determine magnetic north, and act as a compass. This system can also be used to detect nearby ferrous objects.

- **Orientation.** Both Android and iOS APIs provide a method of extracting the global orientation of the phone from the above sensors. Measurements of this abstract sensor are often recorded for convenience, although it does not provide any additional information beyond the above.

Battery Status

The remaining energy level, temperature, and voltage of the battery can be measured. Whether the phone is charging and, if so, what source it is charging from (i.e., wall or USB) can be identified. This is used to help infer whether the phone is with the participant, model power consumption, and to help determine activity (e.g., if the phone is plugged into the wall, the participant is not likely to be walking, and is less likely to be carrying the phone on their person.)

Biometric Sensors

Sensing parameters related to the user's physical status is important for understanding their health, activity, and emotional reaction to the environment. These sensors are typically off-device and communicate with the smartphone via Bluetooth. Both implemented and implementable-in-the-short-term sensors are described here.

- **Weight.** The weight of an individual has important health and psychological implications. Members of my research group have modified a Wii Balance board to act as a Bluetooth scale in conjunction with the phone.

- **EEG.** Electroencephalograms (EEGs) measure electrical activity in the brain. Simple headband-style systems are commercially available and can be used as crude but continuous metrics for measuring focus while navigating through the world, which is of interest to architects and geographers.

- **Heart rate.** Smart watches typically estimate heart rate, which is useful for estimating activity level in health applications.

- **Skin Temperature.** Skin temperature can also be used to aid in estimating calories burned in health applications.

- **Activity Disambiguation.** Watches also contain accelerometers and gyroscopes, providing additional information about wrist movements to complement the movement data from the phone.

On-Phone Surveys

While sensors can provide a great deal of information regarding the physical state of a participant, their mental state must be probed using more traditional survey-based techniques. iEpi provides a framework for deploying short on-phone questionnaires while the participant is involved in data collection. These questionnaires can provide critical insight into why and with whom a participant was engaging in a particular behaviour.

The on-phone questionnaire follows a typical digital survey structure, permitting multiple choice, selection, and free-form questions organized in linear, branching or looping question constructs. This tool captures most of the typical survey question styles, such as single choice selections, select-all-that-apply questions, and short freeform queries. Because questionnaires generally interrupt people during the course of their day, they are typically kept short, to less than five questions. Because typing on smartphones can be difficult, freeform questions are minimized. Questionnaires can be triggered in three ways: on a regular schedule (e.g., every day at 13:15), on a pseudo-randomized schedule (e.g., at least once per day between 9:00 and 17:00), or contextually. Context-specific questionnaires are triggered when a researcher-defined set of sensed conditions are true. For example, if a participant has been detected inside an area known to be a park for a certain amount of time with continuous activity, the phone might trigger a survey asking if they were exercising and – if so – for what reason. Contextual surveys can also be limited by time or frequency (e.g., issue questionnaire if exercising in a park between 15:00 and 20:00, but no more than once per day).

Other Apps

iEpi can work with other apps, providing upload services, and – to properly configured subscribing apps (like the survey tool) – temporally limited contextual information. Only apps configured to operate securely with iEpi can obtain this data. Federating other apps with iEpi can result in new, richer data analysis acquisition, or can close the loop between sensing and action.

3.2.2 Foreseeable and Realized Direct Data Derivations

The data collected by iEpi can be used to directly derive non-intuitive properties about the user and their environment. These derivative uses are only as powerful as the data that is used to generate them and are generally imperfect estimates. A subset of these derivations are listed here, as the potential for collected data is enormous in scope.

- **Nearby Individuals.** Bluetooth can be used to estimate the number of nearby individuals if their personal devices (e.g., phones, headsets, laptops) are in "discoverable mode."
- **Non-Participant Tracking.** Non-participants can be uniquely identified through Bluetooth information. By combining the participant's location with meeting times with identified non-participants, the non-participants can be loosely tracked; the more times a participant encounters a non-participant, the more clearly a non-participant can be tracked, as more data points become available.
- **Frequented / Sensitive Area Detection.** Given enough location data, iEpi can be used to infer the homes, workplaces, and other frequented locations of participants. WiFi and Bluetooth signals can also be used to infer this information.
- **Activity Levels, Activity Type, and Gait.** Accelerometer, gyroscope, and orientation data can be used to infer movement of the phone. If this data is used as a proxy for participant movement, it can then be used to gauge participant activity level and, in some cases, infer the activities in which the participant is taking part.
- **Participant Schedules.** The schedules of participants can be identified by analyzing movement patterns and/or beacons (e.g., known WiFi routers) detected at specific locations at specific times.
- **Phone On/Off Person.** iEpi, through battery data, can detect if the phone is likely off of the participant's person. Combined with accelerometer data, the phone demonstrates high promise in being able to detect if it is likely on the participant's person.
- **Geofence Entry, Exit.** iEpi can determine if the user has entered or exited a precise geographical area, (e.g., "the bowl" at the University of Saskatchewan, a fast food restaurant, a park, or a hotel in Dubai India, etc.). Unlike most context recognition currently performed by iEpi, this can be determined in real time rather than after the event has occurred.
- **Start/Stop Movement.** iEpi can determine if the user has started moving, remained moving, stopped moving, or remained stopped as long as location data is being collected. Unlike most context recognition currently performed by iEpi, this can be determined in real time rather than after the event has occurred.

3.2.3 Actuation Capabilities

iEpi can perform several actions that do not directly collect data:

Surveys. iEpi can issue custom questionnaires through a tool called SurveyApp. SurveyApp can issue most survey instruments that can be represented by buttons, radio buttons, checkboxes, and text fields.

Data Upload. iEpi can upload data to servers. This is not restricted to the data that iEpi generates, such as accelerometer and survey data: any system that places data into its on-phone upload directories – e.g., companion apps such as PhotoFoodDiary – can piggyback on these upload functions.

System Update. iEpi can have its configuration updated mid-study. For example, Bluetooth data collection can be added at any time during an experimental deployment.

3.2.4 Security Capabilities

iEpi has important security capabilities that must be understood to appreciate the security of data it collects and retains. A security audit should be made of the system if it is to be used on a large population outside of an academic environment.

RSA / AES Encryption. Virtually all data in iEpi is encrypted by default via a two-layer RSA and AES encryption scheme. AES encrypts data; RSA encrypts the insecure keys used by AES. Once encrypted in this fashion, only the holder of the RSA decryption key – typically stored on-server – can decrypt this data.

Basic User Anonymization. iEpi, by default, hashes the device's identifier when associating it with data. This is to prevent casual off-device association between a device and the data it contains. Other fields that can directly identify a user, e.g., Bluetooth device name, are either not collected or given an anonymized value.

3.2.5 Scheduling Capabilities

iEpi is currently capable of running scheduling algorithms that do not require data feedback. For instance, a schedule could be created that ran aggressively from 9:00am-5:00pm, but could not depend on data generated by the user to make a decision on how aggressively it should run. Existing scheduling algorithms are listed below, but – in principle – any schedule that can operate without data feedback can be generated:

Duty Cycle. iEpi currently runs most tasks on a duty cycled schedule. This means that the device records a burst of data, sleeps without recording data for a given period, and repeats. Generally, iEpi is used to collect a small data sample (e.g., two minutes worth of data) every few minutes.

Continuous. iEpi is capable of collecting a continuous stream of data from any sensor from the moment it starts to the moment it is killed or the phone dies.

3.2.6 Limited Context Recognition

iEpi has limited capabilities to understand the context in which it is running. Unless specifically engineered into a data processing task (described in chapter 4), context recognition – due to current architectural limitations – can usually only occur *after* the moment of interest has occurred. Contextual information is typically used to limit the number of surveys issued in a day, to make it appear that these surveys are being issued randomly, or to ensure that they are only used at times of interest. The system can also use these capabilities to more opportunistically (i.e., more aggressively) upload data if the phone is being charged. Past experiments have used this capability to issue surveys only when nearby participants were detected. Newer facilities to determine context exactly when data is acquired are currently under development, but are limited to select studies.

3.2.7 Deprecated / Single Study Capabilities

iEpi has periodically contained features used only in a single study or that are not contained in the main iEpi codebase.

EEG Scans. EEG (informally, brainwave scanning capabilities) were implemented for use in the department's HCI lab.

Bluetooth Weight Scale Interaction. Early iEpi incarnations were capable of interacting with Wii Balance boards for the purpose of recording weight.

Precise Participant Identification. Studies involving Exflu (see chapter 4) at the University of Michigan had the capability to match specific phones to participant names, e.g., Phone zz123767j may have been mapped to John Smith, and this mapping was known to all devices in the study. Participant names were stored in an obfuscated fashion, i.e., it was difficult – but not impossible – to reveal the names of participants if the obfuscation algorithm was not known.

Photos. iEpi can be altered to record photos. Introduction of code from auxiliary projects constructed by author can also allow augmented reality views of the world to be constructed. For example, objects can be recoloured to improve identification by those with colour-vision deficiencies.

Diabetes Management (Blood glucose, Weight, Insulin, and Meal Data). iEpi was once used to record diabetes-related data using survey forms.

QR Code / Barcode Scanning. iEpi is easily integrated with barcode scanning applications, which can be used to track real-world objects or tagged individuals (e.g., a participant with a nametag equipped with a barcode).

3.3 Autoparser

As illustrated in the previous section, iEpi produces a wide variety of encrypted, compressed data in vast quantities.[2] Before this compressed and encrypted data can be analyzed off-device, it must be processed and stored. A full understanding of iEpi therefore requires an understanding of the storage process.

iEpi traditionally used a Java-based parser to decipher and store data. The parser was fragile, study-specific, and intimately tied to the specific types of data that iEpi produced. For each new study, a new database needed to be manually built to receive its data. Cross-database incompatibilities and rapidly evolving needs further exacerbated versioning difficulties for maintainers. A replacement was required in the course of this thesis work. As this component has not yet been documented in any formal publication and as it was a significant engineering exercise with implications for the future development of the system, it is described here.

3.3.1 Goals

The parser's replacement was developed with several goals in mind:

- **Table Generation.** Database tables should be automatically generated to match the data produced by a study.
- **Database Agnosticism.** The parser should be able to handle a variety of database types and should do so with minimal intervention from the user.
- **Easy Configuration.** Users of the parser should be able to quickly and easily configure it to match their particular database, study, and data needs.
- **Decoupled Configuration.** The parser should be configurable from outside of the tool, i.e., it should require no recompilation if a change needs to be made to the database address, table, user, or database password.
- **Avoid Java.** Java is clumsy, cumbersome, and slow to use when interacting with databases, yet the legacy parser was built in this language due to dependencies on iEpi decryption and decoding libraries. New languages exist that can interface with the JVM – the virtual machine on which Java operates – so the best alternative should be selected.

[2]For example, SHED1 [39] produced over 45 million raw data records. Comparably large data sets were produced by other studies detailed in chapter 4

- **Minimal Recompilation.** The legacy parser was notorious for needing code augmentation to work with new deployments. This created a versioning nightmare that still causes problems when data must be parsed in legacy studies. As such, the parser should require minimal code changes between studies.

Due to a focus on doing things automatically, easily, and without user intervention, the newest version of the parser has been affectionately termed the Autoparser.

3.3.2 Implementation

The Autoparser is implemented in Scala. This was done for several reasons. Scala interfaces with Java, which is required to parse iEpi data due to its dependence on Java-based encryption and encoding libraries. Scala is clean, easy to write, and can express concepts much more succinctly than Java. It is suited to reading and manipulating configuration files, as the language has built-in XML support, manipulation, and validation facilities. The Autoparser was constructed in approximately a week, yet has proved to be significantly more reliable and – in my opinion – user-friendly than its predecessor.

3.3.3 Configuration

The Autoparser is configured through the use of an XML file. The XML configuration file has three major divisions:

- **Database Configuration.** Contains the database URL, database name, database type (e.g., MySql), and database credentials. Database credentials can also be specified at runtime for security reasons.

- **Processing Configuration.** Contains the folders wherein data to be parsed is stored and where it is to be stored once processed, as well as decryption keys.

- **Data Connections.** Describes how data files are to be decomposed, processed, and placed into the database.

Data Connections

The most powerful abstraction in the Autoparser is the data connection. A data connection represents the mapping between a data file (e.g., iEpi data, survey responses), its fields, and the table and columns of the database into which it will be placed. For each data file to be parsed, the user specifies its type (e.g., iEpi JSON data), its identifying label (e.g., `gps`), its destination table (e.g., `dbo.location`), and its fields; for each field, the user specifies the field name (e.g., `latitude`), the destination column (e.g., `approximateLatitude`), and the destination database type (e.g., `double`). This mapping allows the Autoparser to make table creation statements for all known data types, take any recognizable input data and map it to one of these tables, and identify collected data for which there is no specified storage location.

Implicit iEpi Data Connections

All logger-related data contains the time at which the data was created, whether or not the data is concealed (i.e., marked as private by the user), and the identifier of the phone that generated the data. These fields do not need to be specified by the user in the XML configuration file, as they are assumed to exist in every iEpi generated data file.

3.3.4 Parsing Data

The Autoparser performs several steps when it is instructed to parse incoming data:

1. **Configuration File Validation.** The configuration file is read and basic configuration validation is performed.
2. **Database Connection.** The database is contacted and prepared for interaction.
3. **Database Creation.** If the user has requested that the database's tables be created, the necessary tables are derived from data connections and placed in the database.
4. **Parsing.** Each known iEpi data file (i.e., dat.iepi and survey response XML files) is acquired and parsed. In principle, the Autoparser can be extended to support new data files whose fields are easily extracted; as it currently stands, only known iEpi data files are identified and parsed.

Parsing is a complex step, as each data type has its own decryption, decompression, and deciphering needs. The Autoparser, by default, provides support for two data types: iEpi JSON files and SurveyApp response files. These files are identified by pattern matching and exploit information derived from data connections to extract, decipher, and/or decrypt required fields and information. Once processed, the information to be inserted into the database is passed to an adapter for the appropriate database type. These adapters determine how to translate declared destination types into actual database types, how to connect to the target database, and what implicit fields are required to adequately represent mandatory data.

The Autoparser is not necessarily restricted to the two aforementioned data types and a small set of databases. With refactoring, more data types and database adapters could easily be created and integrated into the system.

3.4 Limitations

The Autoparser cannot readily be instructed to generate derivative data as files are being parsed. For instance, the parser cannot be instructed to generate a duty cycle ID through its configuration file. This must be added to the Autoparser manually when desired or via triggers on the database side. The parser is not parallelized, and may inadvertently process files that are in the process of being uploaded. It cannot update table columns and names in the database when the configuration file is altered; this must be done manually, but such functionality is possible. The Autoparser operates in a single thread, but could be easily parallelized. This will require file locking. The Autoparser should be equipped with mechanisms such that it cannot accidentally grab and process files that are in the middle of being uploaded to the server on which it runs.

Chapter 4
iEpi's Data Collection Engine: Architecture and Use Cases

The text upon which this chapter was based appears as "A Field-Validated Architecture for the Collection of Health-Relevant Behavioral Data" (Knowles, D., Stanley, K., Osgood, N.) in ICHI 2014 [49]. It has been modified to increase clarity, expand terse explanations, and remove text not suited to this chapter.

4.1 Introduction

As noted in earlier chapters, diaries and surveys have traditionally been used to glean insight into the daily activities, motivations, and thoughts of individuals [23, 26, 84]. While these tools generate important insight, they have significant drawbacks [23]. Given the frequency with which these drawbacks are noted within the medical, sociological, public health, psychological, and even geographical [84] literature, the development of a more reliable solution seems desirable. Special-purpose sensor hardware, once employed in the study of structural integrity, animal populations [93], and environmental phenomena [92], have been adapted for use in research into human behaviour [40, 45]. These have shown promise to reduce the difficulties experienced when using traditional tools, but do not address all problems faced by researchers. For example, sensor hardware can be left behind, fail [93], be easily destroyed [26, 40], and can be obtrusive and disrupt normal participant behaviour [84].

Leveraging existing electronic monitoring systems already carried by individuals would alleviate many of the drawbacks of custom solutions while offering the benefits of continuous and low-maintenance data collection. Smartphones, which are now equipped with many sensor types, can be used to capture rich, epidemiologically-relevant data on their owners and populations [1, 23, 38, 39]. Between 47% and 55% of North Americans own these devices, meaning that there is a substantial population in possession of hardware with experimentally useful sensing capacity [16, 82]. Smartphones have the significant added benefit of allowing questionnaires and on-screen forms and user interfaces to be issued [1, 38, 39].

The potential of smartphone-based behavioural monitoring has prompted many individuals to explore the field [1, 38, 39, 57, 64, 77]. Few of these efforts, however, report moving beyond a prototype stage, see use in studies other than those for which they were originally designed, or have been validated by external parties. Proof-of-concept systems are typically difficult to extend, modify, and generalize and are prone to bugs [47].

Tools without clear, concrete plans for expansion are difficult to apply outside of their intended domains. Existing systems with plans for long-term use [1, 64] have either only reported use and evaluation within-group [64] or do not appear to support the sort of perpetual, ubiquitous data collection that many studies require [1]. The proliferation of systems presents many attractive ideas, but there is a lack of contributed, real-world examples where they have been used. While some organizations aim to reduce development barriers in the health sphere [17] and others are developing related proprietary tools [36], there appears to be no example of a standardized, well-travelled approach for health studies with fine-grained data collection needs.

These problems of extensibility, generality, and verifiability have been addressed through the introduction, use, and continual development of a system named iEpi [38, 39] based on a core architecture that facilitates easy extension, testing, and subcomponent reuse. In this chapter, I present the on-device architecture that permits data collection and management across such diverse fields, and explain how low-level architectural decisions impact the utility, flexibility, generality, and robustness of the subsequent tool. I describe several architectural features and how they have been exploited to solve a variety of problems poorly addressed by existing tools. The value of these advances is demonstrated through case studies of the system's use, either previously published or unpublished work with the permission of the authors [24, 25, 38, 39, 77, 78, 88].

4.2 iEpi's Smartphone Data Collection and On-Device Analysis Engine: "The Logger"

When designing a general-purpose architecture for health and behavioural sciences, it is important to address a broad set of use cases, their data collection needs, and their critical data fidelity parameters. In the case of contagious disease, identifying contact patterns with people and locations are of paramount importance. To assess these patterns at a level of fidelity appropriate to each disease, collection regimes and methods must be tailored to the spread and persistence of the pathogen under study. Faster spreading diseases such as influenza may require high fidelity contact tracing of all individuals and locations using microtelemetry data; diseases associated with long-latency periods, such as tuberculosis, may require an increased focus on data from questionnaires triggered by frequent or prolonged contacts made by the participant. If studying drivers of unhealthy lifestyle such as obesity or smoking, then identifying locations, communities of influence, and activity levels are important. Each of these facets may need data collection rates independent of one another. For example, frequently assessing activity level, activity type, and places of activity may be important when studying obesity; community identification may require less aggressive regimes given that influential individuals are likely to be near the participant on a regular basis. Studying environmental exposure to toxins or waterborne illness would require high-fidelity location information, but perhaps no other sensor data. All of these cases could benefit from realtime, case-specific, contextually issued survey instruments for momentary ecological assessment, and potentially different off-device federated sensors, such as participant temperature for respiratory disease, participant weight for obesity, and local meteorological conditions for environmental pathogens.

From this broad use case analysis, it is apparent that an architecture should possess the potential for rapid reconfiguration of sensor loadout and measurement regime, and the capacity to deploy arbitrary survey instruments to participants when contextual cues deem it necessary. Additionally, the capacity for automated and component-isolated testing is required to verify novel configurations prior to study launch. Given that use cases within an experiment may change if conditions change (for example, if an outbreak of a particular pathogen occurs during the collection period, or if participant compliance proves lower than expected), the measurement regime should be adjustable either by remote command or by on-device adaptation.

The system, informally known as "the logger" and depicted in Figure 4.1, has been designed to address the aforementioned use cases. To accomplish this, it possesses an architecture inspired by the pipeline paradigm successfully deployed in computer operating systems [94] and compilers [33]. While this approach has been employed before for health applications [54, 57], the logger's implementation is unique as its pipelines are highly modular, can be reassembled and reconfigured through a specialized programming language, can be quickly crafted and integrated into the system to accommodate new sensing needs, and are deployed on-device. These features facilitate the kind of flexibility of deployment and compartmentalization of testing required for intelligent, reliable data collection independent of specialized server software. This pipelined paradigm entails the design and exploitation of small, extremely focused tools, known in the logger vocabulary as tasks, that are joined together in flexible arrangements, dubbed task chains, to accomplish larger pieces of work. Task chains are invoked on predetermined schedules but only perform work under developer-specified conditions. The behaviour of each task being executed can be modified using task-specific parameters. A high-level view of this can be seen in Figure 4.1.

The pipeline paradigm is valuable in the design of health-centric data collection tools primarily for its focus on modular design and ease of composition of such modules. Modularity enables separation of concerns, simpler blocks of logic, and decoupling – a reduction in the number of dependencies between logical units of code. Decoupling and separation of concerns are, in general, valuable to any software system. They are especially valuable in health-related systems, as their goals can be multi-faceted and therefore difficult to achieve without a planned, intelligent breakdown of the work to be performed. Specifically, studying or identifying nebulous concepts, such as the behavioural impacts of a user's core friend group, the effect of proximity to fast food restaurants on an individual's diet, and the impact of on-device notifications triggered by identifying unhealthy habits, would appear difficult to address without first breaking each task into smaller, more manageable pieces.

Decoupling has the added benefit of facilitating the reuse and recombination of components. For example, a social network inference engine developed for tracking influenza transmission might also be used in tracking the contagion of health behaviours through social influence. Modularity allows for the creation of highly automated test suites and quality assurance regimens that can be extremely difficult to implement in larger,

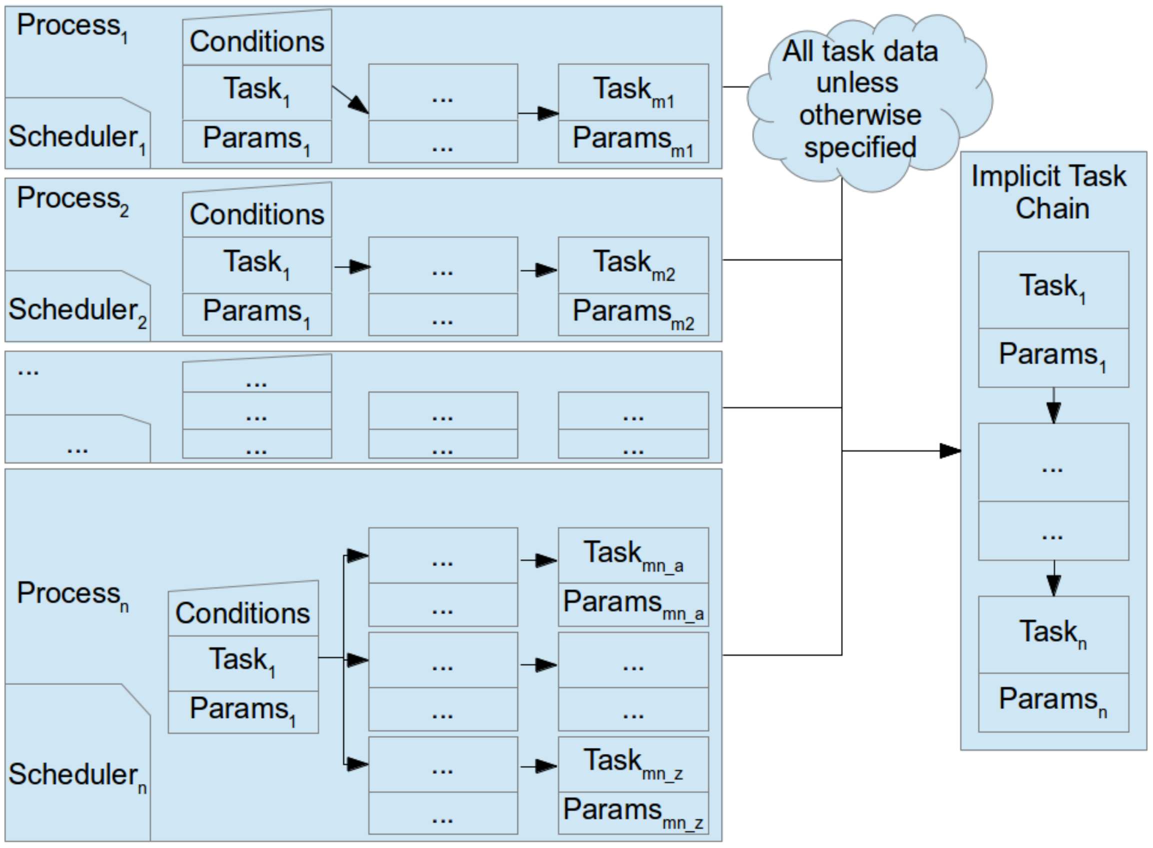

Figure 4.1: A system diagram of the logger. Task chains are contained within processes, each of which runs on its own schedule. Data produced by one task in a task chain is passed to the tasks immediately dependent on it. All output produced by tasks is funnelled into the implicit task chain unless otherwise specified, which encrypts, compresses, and stores data for later analysis. Tasks can output data to many other tasks, which is depicted by Process$_n$.

tightly coupled, and more complex systems, facilitating validation consistent with the stringent requirements of the health science community. The pipeline architecture maintains the additional capacity to easily compose simpler modules into larger structures that accomplish more complex tasks, and fulfills, as a design paradigm, many of the requirements laid out in the general use case.

4.3 Detailed Architecture

For economy of representation, the logger employs a small number of abstractions which can be composed, recombined, and reused to produce complex behaviour found in systems with more specialized components. This abstraction allows for simpler design and easier problem analysis, compartmentalization of concerns, and can produce sophisticated behaviours, particularly when merged with a pipeline design.

4.3.1 Overview

The logger uses several abstractions, shown in aggregate in Figure 4.1. The foundation of the system is a series of concurrent processes that are invisible to the developer. Each process controls the execution of a pipeline of *tasks*, known as a *task chain.* To manage power use in a smartphone-based system, each process is controlled by a user-defined *scheduler*, which wakes the appropriate task chain on a schedule. The head of the task chain checks a series of internal *conditions*, and continues execution only if those conditions hold. Data produced by the head of a task chain is passed through each subsequent task for processing. This may include tasks such as filtering and data transformation, which – in branching task chains – can be performed in parallel. All task output is sent by default to an *implicit task chain* for encryption, compression, and storage. Data that reaches the implicit task chain resides on the device until uploaded by another task or manually removed. A collection of task chains, not shown in Figure 4.1 due to their short-lived nature, can be executed before normal operations begin, which allows critical facilities, such as encryption, to be initialized.

4.3.2 Entities

Task & Parameters

Tasks represent blocks of work to be performed. They can vary enormously in purpose, size, complexity, and degree of reusability, but generally represent a step in a larger data processing endeavour. For example, previous studies employing the logger have visualized participant movement patterns [77, 78]. This endeavour required several steps, including the removal of low-accuracy location data, applying a Kalman filter to improve location estimates, and visualizing the participants' behaviours via a specialized tool. An implementation of this behaviour in the logger might therefore consist of the tasks `(1) sample GPS data`, `(2) filter low accuracy data points`, `(3) apply Kalman filter`, `(4) snap data to map paths`, `(5)impute gaps in location data by assuming shortest path travel`, and `(6) save data in Google Earth format`. New tasks can be introduced into the system with relative ease due to the task framework's modular design. Tasks can perform work without interacting with other tasks and/or data sources, generate data, process incoming data, and any combination of these options if so supported by the implementation. Applying the principle of abstraction by parameterization [55], tasks are further augmented through the use of developer-defined *parameters. Parameters* can be used to set critical or optional values in tasks. Examples include data caps, distance thresholds for movement detection, and encryption key values. Tasks may either take a standalone role (i.e., they produce data for a task chain or independently perform some unit of work) or a dependent role (i.e., they require input from some other task, when used in a task chain). Examples of tasks are given in Table 4.1.

Table 4.1: Examples of existing task types.

Name	**Type**	**Description**
Data Collection Task	Standalone	Samples data from on-device hardware components, including GPS, accelerometer, magnetometer, WiFi, and battery.
Upload Task	Standalone	Uploads stored data to a given URL.
Update Task	Standalone	Updates the logger's configuration from a given URL.
Database Managing Task	Standalone	Pushes in-memory data to the logger's main database in batch operations to improve storage efficiency.
Survey Task	Standalone	Shows a survey specified by the developer.
Encryption Configuration Task	Setup	A setup task that establishes the RSA encryption keys used by the logger.
Anonymization Configuration Task	Setup	A setup task that initializes basic data anonymization facilities.
Place Identification Task	Dependent	Identifies a place label from location data, e.g., (52.109542, -106.637590) produces the label *Wiggins Park, Saskatoon, Canada.*
Place-Triggered Survey Task	Dependent	Triggers a survey if the current place is one of interest.
Compute Average Task	Dependent	Computes the average on a set of incoming data. If the data type cannot compute an average, this task will reject the data.

Conditions

Like tasks, *conditions* are blocks of logic that can be created by the programmer; in contrast to tasks, their invocation produces a boolean value rather than streams of data. In line with the Specification design pattern [32], conditions can be linked using AND, OR, and NOT constructs to form more complex predicates. If the boolean expression attached to a task returns false, it will not be executed.

Conditions were traditionally based on a simple central cache, but the integration of advanced condition monitoring techniques devised by other authors such as Kang [45] could provide an efficient way to improve the existing system. More recent studies have already moved in this direction by employing direct task-to-condition connections. This emulates Kang's work and the work of Pathak [75, 76] at a rudimentary level, directly funneling data to conditions that require it. This approach has shown a great deal of success as it enormously simplifies the identification of situations of interest. Specifically, conditions get only the data they want and can use, exactly when it is generated, and can immediately compute a result without needing to store unnecessary data. The caching approach, in contrast, forces conditions to operate on old data, query for the data they require, make assumptions about what constitutes a useable "burst" of data, and requires significant memory overhead, as all data must be stored in a fast – and typically in-memory – database. Examples of existing conditions can be found in Table 4.2.

Table 4.2: Examples of existing conditions.

Name	**Description**
Is Plugged Condition	Indicates if the phone is plugged in to a power source; useful when determining if power-intensive actions should be undertaken.
Hour Range Condition	Indicates if the current time is in the half open range [start hour, end hour).
Daily Limit Condition	Used to prevent some action from being executed more than a set number of times.
Has Stopped Condition	Indicates if an individual has stopped moving.
Nearby Participant Condition	Indicates if a known participant has been detected nearby recently. The time threshold defining "recently" is provided as a parameter.
Location Changed Condition	Indicates if an individual has changed location, e.g., moved further than some threshold since the condition was last checked.

Task Chains

Tasks can be linked into *chains* to facilitate pipelined processing. They are invoked as a single unit. This is a useful abstraction for staged data processing, logical decomposition, and code reusability. Tasks can be connected together such that there is a one to many relationship between them, i.e., one task can produce output that is fed to many other tasks. This is an unofficial feature, as it is not exposed by iEpi's configuration language Seddacco (chapter 5), it is not exhaustedly tested, and task concurrency limitations (chapter 7) cause bottlenecks in this process. Small infrastructure changes could also support many-to-many relationships between tasks in task chains. Such changes, however, would increase the burden on task and condition developers, as they would need to accommodate the possibility for data arriving from many sources, from many bursts, and from data streams with different accuracies and granularities.

A potentially simple resolution to this issue would involve tasks and conditions declaring the number of sources from which they could receive data. For example, iEpi's central data receiving instance could be marked as a receiver of data from an infinite number of sources; activity detection tasks could declare that they only accept data from a single source to simplify data processing. An additional issue arises from the issue of cyclic data processing that could form unintended infinite loops. Other systems avoid this pitfall through the use of directed acyclic graphs. iEpi could enforce this policy through Seddacco, as it is used for virtually all logger configuration. Existing task chain examples are given in Table 4.3.

Schedules

Task chains are invoked on developer-defined *schedules*. While support exists for a variety of schedule types, e.g., adaptive sampling based on historical user behaviour and data transfer regimens based on the time of day, the default option is a duty cycle-based execution regime. Highly specialized schedules have been introduced for automated test cases, however, meaning that the introduction of new, more sophisticated applicable schedules is both possible and readily achieved. Existing schedules operate by exploiting Android's alarm service [4].

Table 4.3: Examples of existing and hypothetical task chains.

Concept	Tasks Sequence	Explanation
Location Entry Detection	Collect GPS data, convert GPS data to location data, determine if current location has changed	Periodically collect GPS data, determine if the data corresponds to a known location label, then determine if the location label has changed. Employed in conjunction with foreign collaborators.
Stop/Start Detection	Collect GPS data, convert GPS data to location data, compute location average, determine if current and previous location averages are a significant distance apart, use statechart to identify motion class.	Take a periodic snapshot of location data and compare with the previous snapshot. Determine the distance between these locations. Consult a statechart to distinguish between starting, stopping, moving and resting. Employed in conjunction with foreign collaborators.

Recently, Seddacco has been augmented to allow user-specified, non-duty cycled schedule types. As such, new schedules can be written in iEpi and directly used by the system's end users. A current architecture limitation, discussed in chapter 7, prevents these schedules from accessing data that their respective task chains produce.

Implicit Task Chain

The *implicit task chain* encompasses several tasks that are almost always necessary for data collection, such as data encryption and local storage. The implicit task chain does not run on a schedule; rather, it is a global entity that is connected by default to the data output channels of all tasks in the system. The result is that virtually all data producing tasks will have their data encrypted and retained for later review without explicitly specifying that these operations be performed. The steps in this process can also be seen in Table 4.4.

Steps in the implicit task chain are not currently tasks. Rather, they are static entities fitted to the task interface to accommodate legacy constructs. Task chains are not aware of their existence beyond this interface. This would allow iEpi to swap out the data logger with any other entity, e.g., a true task chain, at any point, and would even allow the user to specify the entities within the implicit task chain via Seddacco. This would be extremely useful for studies that, for example, needed specialized anonymization support or selective data storage and upload policies.

4.4 Case Studies

From the existing literature, similar ubiquitous data collection systems in the non-commercial sphere [48, 54, 57, 64] have only been used in the context of a single experiment or have not reported moving beyond the prototype stage; proprietary systems such as [36] appear specific in their purpose and do not appear

Table 4.4: Steps in the implicit task chain.

Task	Description
Cache Data	Cache data locally for use by conditions and tasks. Cached data is unencrypted so it can be read, but is typically stored for a short period. This is used infrequently in modern versions of iEpi, but used regularly for study-specific conditions in legacy versions.
Filter Privacy-Sensitive Data	If the user has enabled privacy mode, strip all data that has not been explicitly coded for retention. This filtration, if enabled, will generally retain only timestamp, device, and data type information: this allows users to keep their activities private while allowing researchers to differentiate between data loss due to device failure and data loss due to user request.
Encrypt Data	Encrypt data using the RSA asymmetric algorithm for later retrieval or transfer.
Store Data	Store data in a database; when the database becomes sufficiently large, data is compressed to file for upload. Files are batched by an external task for efficiency.

open for use in external spaces. In contrast, iEpi has been deployed in several different pilot experiments, ranging from monitoring behaviour for understanding the spread of contagious disease, to providing data to persuasive health games. In this section, those studies where the use of iEpi has either been published or where permission has been obtained to disclose overall experimental parameters prior to publication are highlighted. Additional studies have been run using iEpi, but collaborating researchers have indicated that they do not wish to disclose its use prior to the publication of their results.

4.4.1 Communicable Disease

Communicable diseases, such as influenza, Norovirus, and the common cold can adversely affect public health. These diseases are spread by person-to-person contact and contact with contaminated locales. Understanding individual contact patterns and mobility can help trace and combat the spread of these illnesses. Traditional tools such as surveys cannot always capture this information at sufficient temporal fidelity. iEpi has been used to collect human behavioural data over three flu seasons in experiments named Saskatchewan Human Ethology Dataset 1 (SHED1) [39], SHED2 [38], and Exflu, summarized in Table 4.5. SHED1 was a proof of concept study to demonstrate the uses of smartphone-derived microtelemetry data. The collected data permitted the creation of investigatory tools, ranging from simple queries to agent-based simulations of flu transmission. SHED2 sought to take SHED1 one step further by integrating all existing iEpi components with external tools and algorithms. Exflu was designed to determine the effectiveness of social distancing in reducing the transmission of influenza.

One of the primary reasons for developing the logger was to study the relation between individual behaviour and pathogen transmission using microtelemetry data. SHED1 demonstrated that even a relatively simple task-based architecture could be used to facilitate microtelemetry data collection. SHED2 demonstrated that such an architecture could be integrated with off-device tools and other on-device apps to measure context not immediately available from on-device sensors. Exflu demonstrated that some of this work could be performed within the architecture itself; through the introduction of caching mechanisms and participant-identification modules, contact recognition – normally done off-device – could be done in near real time. This recognition could further be used to affect other tasks and device behaviour through the issuance of contextually-aware surveys.

Table 4.5: Communicable Disease Datasets Collected Using iEpi

SHED1 [39]	
Demographics	39 participants, over 45 million raw data records
Duration	Five weeks
Purpose	Use detailed microtelemetry data to build an agent-based simulation model of a flu-like illness amongst a real population.
Data Collected	Accelerometer (one minute burst), bluetooth (one minute burst), WiFi (three second burst), GPS (two minute burst), and battery data (10 data points) was collected every five minutes.
Outcomes	Mobility and contact dynamics were extracted and place analyses – e.g., identifying the most frequented locations – were performed. Simulations of disease transmission were produced by replicating the habits of individuals while changing disease dynamics.
Realized Architecture Benefits	The modularity of the data collection task allowed data extraction from diverse data collection sources; the task could extract data from any source fit to the prerequisite stream interface. Modularity allowed quick identification and localized repair of severe battery bugs post-study.
SHED2 [38]	
Demographics	38 participants, over 140 million raw data records
Duration	Total: 16 weeks. As analyzed in reference: 34 days.
Purpose	Employ all facets of the iEpi data collection system in a single study. This included sensors, a survey tool, an external weight scale, photo-based online food diary, indoor localization tools, a data analysis suite, aggregate-level simulations, agent-based simulations, and external standalone tools.
Data Collected	Accelerometer (one minute burst), bluetooth (one minute burst), WiFi (three second burst), GPS (two minute burst), and battery data (10 data points) was collected every five minutes. Questionnaires, PhotoFoodDiary, and bluetooth weight scale data collected as provided by participants.
Outcomes	Data was collected and a basic analysis was performed; analysis was similar to those performed in SHED1 [39]. Results [81] include reproduction of Song's work [86] using a different locationing technology and findings on mobile entropy sensitivity. 1841 survey responses and 425 weight scale readings were acquired.
Realized Architecture Benefits	Other apps can integrate with iEpi's services at a basic level. PhotoFoodDiary exploited the logger's upload capabilities without modification to the system. Contextual questionnaires were introduced through a task that communicated with an external survey tool and used data in a cache connected to the logger's database.
Exflu	
Demographics	103 participants.
Duration	iEpi-facilitated component: Approximately 2.5 months

Purpose	Determine the effectiveness of social distancing in reducing transmission of influenza in a university dormitory setting.
Data Collected	Accelerometer (five second burst), bluetooth (30 second burst), WiFi (10 second burst), and battery data (5 data points) was collected every five minutes. Questionnaires concerning recent interpersonal contact habits were issued at most two times daily at random times.
Outcomes	Confidential until publication.
Realized Architecture Benefits	New conditions were rapidly introduced to identify recent and non-recent contacts. These conditions were thoroughly verified using automated test suites to ensure that they would work correctly in a larger deployment. Tasks were modified to incorporate cached data – e.g., the names of nearby participants – without system-wide changes or significant architecture modification.

4.4.2 Human Mobility

From reducing sedentary behaviour via modifications to the built environment to assessing the impact of environmental media messaging, to understanding exposure to environmental or zoonotic vectors, location is important in assessing health behaviours. While GPS is an obvious sensor choice, it performs poorly in institutional buildings and in densely populated urban areas. Collaboration with geographers was undertaken in two studies – SHED3 [77, 78] and SHED4 – to investigate the use of smartphone-derived data in indoor localization. Results of these experiments are currently being analyzed through simulation on the relative impact of spatial and contact contributions to the spread of contagious disease. SHED3 was a month-long study in which detected WiFi signals were acquired, analyzed, and produced location estimates along indoor routes to create accurate movement traces. The intent was to couple unstructured data sets with GIS techniques to understand building use. SHED4 constitutes a replication of SHED3 to increase sample size and evaluate repeatability. Mobility datasets are summarized in Table 4.6.

The logger was not designed for the purposes of indoor localization. Nevertheless, its architecture proved general enough that it could be used for these studies with only configuration changes. This suggests that the architecture could be repurposed for other studies requiring movement data in either indoor or outdoor environments.

Table 4.6: Mobility Datasets Collected Using iEpi

SHED3 [77]	
Demographics	37 students, over 36 million raw records
Duration	One month
Purpose	Develop a methodology for combining indoor localization systems and data provided by smartphones. Demonstrate that low quality data can be cleaned and used to provide meaningful and accurate movement tracing through post-processing. Demonstrate that tracking data can reveal insights that cannot be revealed by traditional tools.

Data Collected	30 second samples of accelerometer, magnetometer, bluetooth, compass, and WiFi data every two minutes, and 10 battery records every two minutes. Survey and sketch map data were collected from each participant.
Outcomes	Developed a method to provide accurate indoor localization information by combining the logger, SaskEPS, and Walkable CentreLINE. Spaces frequented by individuals that were not reported in surveys and sketch maps were identified, e.g., bus terminal and skywalks between buildings.
Realized Architecture Benefits	The logger's ability to be readily and quickly reconfigured using Seddacco allowed researchers to reuse existing functionality to accomplish the study's data collection needs. Neither additional coding nor recompilation was required.
SHED4	
Demographics	33 participants, over 67 million raw records
Duration	Four weeks
Purpose	Trace individuals through a space while simultaneously monitoring their activity levels for the purposes of classification.
Data Collected	30 second samples of accelerometer, magnetometer, bluetooth, compass, and WiFi data every two minutes, and 10 battery records every two minutes. Data was collected frequently to better assess movement. Survey and sketch map data were collected from each participant.
Outcomes	Confidential until publication.
Realized Architecture Benefits	Configuration designed for SHED3 could be reused with slight modification to suit this new study. An update task was used to reconfigure the logger mid-study.

4.4.3 Gamified Interventions

Effective encouragement healthy behaviour in the general population has become a topic of great interest for both researchers and health policy makers. From elevating physical activity to promoting better eating habits, the health community is increasingly shifting its focus upstream to preventative strategies. Collaborators have used the logger as a base collection and communication mechanism for several gamified interventions, shown in Table 4.7. Gemini 3 [88], the successor to Gemini 1 and 2 [89, 91], was used to study the impact of gamification on activity levels. It augmented a role playing game through the addition of a support character whose strength and abilities depended on the activity habits of the player, measured using a version of the logger, requiring a custom GUI that allowed players to monitor the abilities of their support character. PasswARG [24, 25] was created to reduce the startup costs of simple, educational exergame development. Games were based on an Easter egg hunt mechanic. The logger's role in this endeavour was to act as a silent data collection engine that collected movement and activity data during play sessions for subsequent analysis. The Flu Game was used to study the impact of gamification on illness avoidance. The game used a modified version of the logger to cause a phone to become "ill;" an ill device would audibly sneeze (requiring the addition of an audible user interface) and spread its illness to neighbouring phones. The object of the game was to avoid being infected by others.

Gamification studies have demonstrated that our architecture can be used in situations requiring varying levels of customization. In PasswARG [24, 25], the application simply needed to have its configuration changed; this demonstrates that the architecture is general enough that it can be easily repurposed without recompilation. In Gemini 3 [88] and the Flu Game, the application required GUI modification and custom communication mechanisms. The system can rapidly be retasked to form the basis of new, completely disjoint health applications for which it was not designed.

Table 4.7: Gamified Interventions Enabled by iEpi

Gemini 3 [88]	
Demographics	Two rounds, twelve participants in each round.
Duration	10 days per round
Purpose	Study the impact of gamification on personal health and activity levels. Determine how game-related alerts delivered to the player impact gameplay.
Data Collected	Battery (five seconds), bluetooth (30 seconds), accelerometer (one minute), and WiFi (three seconds) every five minutes.
Outcomes	Confidential until publication.
Realized Architecture Benefits	The architecture's modularity allowed for the introduction of a new GUI and support character interaction system. It also allowed existing components to be removed from their original contexts, modified, and reused elsewhere, e.g., data uploading and transfer code was used to upload Gemini-specific data.
PasswARG [24, 25]	
Demographics	A group of 18-20 students each week for six weeks; students ranged in age from eight to twelve years old.
Duration	Six weeks.
Purpose	Facilitate the analysis of ubiquitous exergames; used iEpi and the logger to identify participant behaviours during play.
Data Collected	Near-continuous accelerometer and GPS data
Outcomes	Primary outcomes demonstrated feasibility of system: students were able to both create levels for other students and complete created levels. Students exercised with the game as the average distance moved by students was 481.5 meters in 15 minutes.
Realized Architecture Benefits	The logger was quickly reconfigured using Seddacco commands. This demonstrates that the logger's architecture is subject-agnostic; a simple tweaking of configuration files can allow it to be repurposed for a variety of uses.
The Flu Game	
Demographics	28 participants
Duration	One week
Purpose	Determine the impact of human behaviour on the spread of pathogens.
Data Collected	Siphoned data from SHED2
Outcomes	Due to the length of the study, no significant results can be drawn.
Realized Architecture Benefits	The game required frequent bidirectional device-server communication. New upload and communication mechanisms were needed to facilitate this functionality. The logger's task architecture facilitated the introduction of Flu Game-specific send/receive communication mechanisms.

4.4.4 Understanding Pathways Linking Health Outcomes to Health Behaviours

Individual health outcomes are often linked to their health behaviour and environment through diverse pathways whose effects are difficult to distinguish. The logger was used to study this subject in two disjoint areas, patient mobility and diabetes management, as shown in Table 4.8. Increasing mobility is a welcome sign in patients rendered immobile by disease and injury, particularly in elderly patients. Manually quantifying the extent of such mobility – and, by extension, elucidating the reciprocal effects by which it is linked to health outcomes – is difficult using traditional instruments. The logger was used to monitor the activity and mobility habits of elderly patients in a hospital setting in conjunction with step- and movement-counting algorithms. Second, gestational diabetes – a form of glucose intolerance first identified during pregnancy – can negatively impact the well-being and health outcomes in both the mother and her child [74]. While great variability is observed in health outcomes for those with gestational diabetes, the specific contributions of various risk factors – such as physical activity, sedentary behaviour, diet, and compliance with pharmaceutical (e.g., insulin, metformin) administration – is poorly understood. Smartphones demonstrate promise for being able to help monitor such factors, and thus foster better understanding of the effects of those risk factors on blood sugar levels and health outcomes. Eventually, such understanding might help contribute to improved understanding of the specific pathways by which environmental and contextual factors and interventions affect gestational diabetes outcomes.

In the hospital setting, the logger underwent no modification and was simply configured for fine-grained, continuous data collection. When deployed to pregnant mothers diagnosed with gestational diabetes, it required a new user interface to accept a variety of new data and further changes to its data storage mechanisms to facilitate the retrieval of potentially months-old data. Because of the modularity of the system, the changes were highly localized: the logger's caching mechanism was swapped out for an implementation that could facilitate data retrieval and the user interface was altered to allow new recording and feedback functionality. Coupled with the ease with which the user interface may be removed and replaced, this flexibility may also facilitate user interaction heavy studies, especially those with visualization and direct feedback needs.

4.4.5 Identifying Human Activity

It is a difficult task to identify the actions an individual is taking from sensor data alone. This identification, however, is valuable in determining user behaviour for the purposes of health monitoring and context detection. iEpi is currently being used to help make this process simpler. An undergraduate student in the DISCUS lab who had never seen nor used iEpi before modified the system such that the user could report their activity type and intensity for a given period. An accompanying, off-device machine learning algorithm was then created to associate this information with accelerometer and other iEpi-derived data. The hope is that this endeavour, detailed Table 4.9, will lead to automatic identification of user activities.

Table 4.8: Health Pathway Experiments Employing iEpi

Patient Mobility	
Demographics	22 elderly patients
Duration	Approximately one month
Purpose	Track the mobility and physical activity level of patients to study their impact on health outcomes.
Data Collected	Accelerometer, WiFi, bluetooth, and Gyroscope data continuously. Phones swapped out by attending staff to counteract battery life reductions induced by heavy data collection regime.
Outcomes	Percentage of time that each participant spent performing a variety of inferred activities, e.g., shifting, fast walking, was identified. Step counts were inferred to quantify exact amount of movement that was undertaken.
Realized Architecture Benefits	Scheduling flexibility allowed for a near-continual data collection regime.
Gestational Diabetes	
Demographics	Ongoing; the final participant intake was in September 2014. Participants are pregnant mothers diagnosed with Gestational Diabetes.
Duration	Varies per participant due to stage in pregnancy at which phone was distributed.
Purpose	Determine the feasibility of studying risk factors for gestational diabetes using smartphone-based data collection.
Data Collected	30 seconds of accelerometer, bluetooth, and WiFi data every five minutes. Five battery records every 5 minutes for compliance estimation. Two and a half minutes of GPS data every 15 minutes for periodic location estimation. Up to six activity surveys issued each day at random times. Participants were asked to manually enter a variety of diabetes-relevant data, such as photos of meals, blood glucose readings, and insulin doses.
Outcomes	Pilot study currently underway.
Realized Architecture Benefits	A new GUI was required, including screens to enter and review diabetes-pertinent information. New data types were added. A new caching system was required to manage health information for personal use. Once complete, new components were either added atop the existing system or simply replaced existing modules; integration into the system was effectively trivial, and testing contained.

Table 4.9: Identifying Activity Types with iEpi and Machine Learning

Activity Type Identification	
Demographics	Various undergraduate students, graduate students, and faculty at the University of Saskatchewan
Duration	Ongoing
Purpose	Collect sensor and self-reported data to help automatically identify activity type via machine learning algorithms.
Data Collected	Accelerometer, Magnetometer, Gyroscope, and Self-Reported Activity Label Data.
Outcomes	Ongoing. Successful identification of phone location in relation to owner, i.e., on- or off-person.
Realized Architecture Benefits	A new GUI was introduced. The GUI produced a novel data type, i.e., activity label data, that was readily integrated into the system. Experimentation occurred wherein data collection streams were introduced outside of the normal task infrastructure. The system was modified to accommodate new schedule types for experimental data collection.

4.5 Discussion

iEpi's logger has facilitated a variety of studies within the health field. These studies include data collection for influenza transmission modelling, monitoring behavioural change when pathogen avoidance is gamified, developing methodologies for creating reliable indoor positioning services, coupling real-world behaviour with role playing games to promote healthy behaviour, monitoring mothers diagnosed with gestational diabetes, observing patient movement and mobility within a hospital ward, and helping to identify participant activity types. The logger's architecture facilitated such studies through a modular, pipeline architecture that allowed novel components to be rapidly introduced, while keeping unit testing constrained. It also facilitated the reuse of existing components for unrelated purposes, e.g., modules originally used for contact detection were used to study the impact of ubiquitous games on human activity and social patterns.

The system's architecture and design addresses many problems identified in the literature:

1. The architecture of the system has been demonstrated to be highly flexible and extensible. This allows it to be used in a variety of disjoint scenarios, facilitating a broad spectrum of health behavioural studies, as demonstrated in the previous section.

2. The system is easily decomposed. This facilitates reuse, ready realization of novel functionality via task composition, separation of concerns, division of responsibilities in development, and testing. Such modularity also facilitates highly detailed automated testing procedures to ensure continual system robustness and unbroken functionality; our automated test suite includes nearly 900 tests used to pinpoint errors introduced during or after any development, extension, or repair procedure. This count does **not** include the additional precondition and postcondition assertions present in almost every single

public method and many non-public methods, which – coincidentally – amounts to over 900 additional safety checkpoints.

3. The system can be outfitted to collect both manual input data, such as blood glucose levels, and background data, such as GPS readings.

4. The system provides facilities required by many studies [48], including encryption, privacy mechanisms, and data module interoperability.

5. The system can be used to emulate traditional tools, such as diaries and survey instruments, but can remind the participant to use them when memories are fresh. It can further use sensor data to verify or calibrate manually supplied data, and self-reported data to help explain and clarify intention, perception and other subjective factors underlying the sensor data.

6. The system solves problems encountered by special purpose hardware and, unlike inconvenient additional systems, is on the participant's person if they are carrying their smartphone.

7. The system, with small modifications, could be deployed in a scalable fashion: given a mechanism with which participants could download a study's configuration, the app could be deployed to the Android app store and downloaded by many participants.

8. The use of an interpreted configuration language allows study organizers to alter the measurement regime according to circumstances by either changing the configuration file on the device or by pushing new configuration files over the network.

In its current state, the system can be used to perform a variety of health-related data collection and processing tasks. Raw sensor data, such as GPS, WiFi scan results, and accelerometer readings, can be collected at user-predefined levels of granularity. This can be used to infer high-level behaviours, such as fast food consumption, activity levels, and spatial-temporal movement behaviours. Traditional forms of data, such as diarying and survey responses, can also be recorded using the system. These can be issued intelligently through the use of conditions and contextual data. Study compliance can be checked on an as-desired basis through the monitoring of incoming data. Participant privacy is protected through the use of encryption and basic anonymization mechanisms. These facilities, as have been shown, can be used for a variety of purposes, including disease transmission modelling and determining the effect of gamified activity interventions.

The ultimate goal is for this system to be widely used by researchers studying spatiotemporal human behaviour. While it has demonstrated its efficacy in a wide range of health applications, improvements and extensions are always possible. The introduction of new functionality is possible via libraries that can be accessed using Seddacco statements. Facilities defined in the literature, such as highly efficient condition checking [45], intelligent and automated interactions with external apps [48, 64], and adaptive sampling regimens [57], could be integrated into the system. Integration with external sensor hardware is currently possible, but could be made simpler. Network-level operators could be introduced to enable population-level, rather than individual-level, investigation and data acquisition.

By integrating additional features, current maintainers of the system aim to extend the logger and iEpi such that these tools can serve as a common platform for many health science applications. They plan to use the system in a larger variety of studies to catalyze, test, and perfect future developments. They further wish to make the system more readily available to other researchers to bring the benefits of my architecture to other academic and non-academic endeavours, and are actively taking steps to make this a reality.

Chapter 5
iEpi's Domain-Specific Language for Node-Level Configuration: Seddacco

The text upon which this chapter was based appears as "Seddacco: An Extensible Language in Support of Mass Collection of Health Behavior Data" (Knowles, D., Stanley, K., Osgood, N.) in HI-KDD 2014 [50]. It has been modified to enhance clarity, expand terse explanations, and remove text not suited to this chapter.

5.1 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. 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 [59], MacroLab [42], and Semantic Streams [100] 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 [64] and the XML-like files of EpiCollect [1].

For brevity and ease of discussion in this chapter, the term *individual-centric mass data collection system* (IMDCS) is used to reference systems that collect large amounts of data on individuals in the context of distributed sensing applications. As previously discussed, these systems can be complex to both configure and use, due to considerations unfamiliar to domain experts, such as concurrency, hot swapping, system configuration updates, power-dependent collection throttling, events, and collection optimization. In the vast majority of such cases, domain experts are unskilled in these areas. Should designers of individual-centric mass data collection systems wish for their IMDCS to be widely used, it is beneficial for languages or tools for configuring such systems to resolve, mitigate, or hide their peculiarities from domain experts. Reducing usage barriers, however, is not always beneficial, as simplifying a system may reduce its flexibility and thus the range of studies for which it can readily be used. An easy-to-use configuration language should therefore still expose enough features of the underlying system to enable a range of experiments. A small list of features common in many systems or plausibly expected by domain experts is given here:

- The system must be energy efficient, support a variety of collection modes, and have communication support. These are fundamental requirements of any wireless sensor network deployment, which encompasses individual-centric mass data collection systems. Without these features, the system will be restricted to niche applications.

- Flexibility and extensibility are key if additional data streams are expected or if protocols may be modified mid-study. Static systems are difficult to modify and roll out – particularly when there are hundreds or thousands of devices employed in a particular deployment. While static systems are easier to develop, they can impose large maintenance costs on domain experts who will sometimes need to alter their studies as data comes in, or undertake staggered rollouts (e.g., to roll out additional surveys).

- The system should facilitate interaction with external devices and applications. Every data collection system known to the author supports on-device sensors, but few interact with external sensors [48, 64]. Domain experts such as health scientists may wish to use external devices such as blood glucose monitors, heart rate monitors, and step counters, or may simply wish to interface with existing, trusted applications. A lack of interoperability is therefore likely to alienate some intended users of individual-centric mass data collection systems.

- The system should capture and respond to events of interest. Many collection systems provide event support through which a system may respond to situations of interest [7]. While events are difficult to program [7], they are, in our experience, natural and intuitive for many experts to talk and think about: users of our our system have repeatedly stated the need to identify entry to some location, contacts with other individuals, moments of increased activity, and so on. This reality makes it seem wise for individual-centric mass data collection systems to facilitate event detection and response.

These general features are not representative of all needs that system designers might address when developing a new collection system, let alone the configuration language required to make them accessible to domain experts and the programmers they might employ. This small list, however, is illustrative of the fundamental problem: there is enormous diversity in the features a configuration language must incorporate. Developing such languages without at least some conception of the major ideas to be captured may result in tools that are unnecessarily restricted.

While it is impossible to capture all features needed by all users in a user-friendly fashion, it is possible to capture the needs of many individuals in a particular domain. This problem is addressed through two contributions. The first is a description of the requirements that a configuration language must meet to be of use in individual-centric mass data collection systems targeted at health applications. The second is a language designed to meet these requirements that interfaces with a tool demonstrated to work in this area.

5.2 Requirements Analysis

Many wireless data collection systems employ macroprogramming languages to simplify the configuration of their constituent collection devices. Three primary components motivate the design of such macroprogramming languages: the underlying, fundamental use case of the collection system that has been or is being designed; the capabilities of the supporting hardware and software; and general guidelines from the domain in which the language will be used. The use case provides a set of features that are needed to meet the functional requirements of the studies to be undertaken. 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, which may introduce syntactic limits and/or abstractions to either work around or work with these constraints.

5.2.1 Fundamental Use Case

Our fundamental use case relates to monitoring human spatial-temporal behaviour. This requires that information be collected on when, why, and with whom individuals were at specific locations. The more accurately that this information can be acquired, the more likely it is that domain experts will be able to draw meaningful conclusions from it.

The task of monitoring behaviours is fundamentally a problem of relating observations of health behaviours (e.g., physical activity, sedentary behaviour, tobacco use, dietary intake) to the set of exposures to which an individual within a population is subjected (i.e., the exposome [13, 101]). These exposures include direct exposures and environmental restrictions that affect an individual's activities; for example, long daily commutes may remove the time needed for physical activity, the nearest grocery store might be far away and increase the likelihood of making poorer food choices, the local neighbourhood may lack sidewalks or require navigating convoluted streets, or an individual may be enmeshed in social networks exhibiting high prevalence of risk behaviours or environmental risk exposure. While eliciting 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 germane to 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.

Sensing an individual's behaviours presents many low-level technological challenges. Spatial monitoring requires both indoor- and outdoor-specific positional telemetry, as well as activity and social proximity sensing; sensing the latter two components generally requires either that special-purpose sensors to be deployed, entails inference from more general sensors, such as accelerometers and Bluetooth signals. The monitoring 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; these streams, even within the same operating system and platform, can have vastly different APIs and must be dealt with on a case-by-case basis. The system must also be able to handle data of a near continuous (e.g., location) and discrete (e.g., questionnaire responses) nature. The granularity of these data streams can vary widely and may need to be collected on a synchronized schedule to be of use to researchers (e.g., so that data fusion may be possible at specific moments in time). The system should be able to conditionally trigger survey instruments (e.g., questionnaires), enable certain sensor modalities, communicate with a researcher's server and/or database, and perform miscellaneous housekeeping duties (e.g., compress data). 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.

5.2.2 Domain Requirements

The motivating example points to many items that an IMDCS should address. A broadly applicable configuration language in this space should accommodate such a system's feature set, and 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, maintenance tasks, etc.

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

Conditional Execution. Facilitate conditional execution based on the monitored participant's state to capture context-sensitive information that would otherwise 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. As static deployments are difficult to alter once released, dynamic reconfiguration is ideal.

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's precise operations – e.g., the method by which accelerometer data is translated into activity detection – be configurable and fine tuned. 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 [62, 87], and general programming languages. Relevant design principles from these literatures include the following:

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.

Decoupling. 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 and ideas.

5.2.3 Platform Requirements

The configuration language named Seddacco (**SE**lf **D**ocumenting **D**ata **A**nalysis and **C**ollection **CO**de) was developed in this work to address the aforementioned requirements. Seddacco interfaces with the iEpi data collection system to implement domain-specific features and tailors its syntax to succinctly interact with iEpi's abstractions (chapter 4).

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 as these are the basic unit of work in iEpi.

Task Parameterization. Seddacco must allow tasks to receive parameters, as many tasks require parameters to operate correctly – e.g., the survey issuing task requires the name of the survey to be issued.

Task Interoperability. Seddacco must permit task chaining for larger data processing constructs. iEpi uses a pipeline-based architecture that encourages small, recombinable tasks; this functionality is lost if Seddacco cannot facilitate task chaining.

Startup Logic. Seddacco must facilitate initialization as some configuration elements – e.g., encryption key generation – must be performed before other tasks begin to generate or use data.

Implicit Processing. Seddacco must relieve the user of the need to repeatedly specify implicit data processing tasks, such as post-collection data encryption and compression.

Abstraction Linkage. Seddacco must allow users to combine iEpi abstractions, and should automate as much of this process as possible [62].

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.

5.3 Seddacco Language Design

Seddacco is an extensible domain-specific language [62, 87] designed to configure individual-centric mass 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 (e.g., collect accelerometer data), execution constraints (e.g., only if outside), and temporal constraints (e.g., in thirty second collection bursts) 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 5.5, is designed to make this process simple, natural to express, and capable of readily capturing domain knowledge. Specifically, the language takes an English-like form, avoids the use of traditional programming symbols such as curly braces, and allows the user to specify data manipulation elements as human-friendly phrases, e.g., `collect heart rate data`, rather than more traditional functions, e.g., `collectData(heartrate)`.

5.3.1 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 [62, 87] to programmer-provided constructs – such as algorithms to be performed and the schedules on which they run – that are manipulated via English-like statements supporting customization and extension on a per-application 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 and iEpi follow 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 and a scripting language. iEpi provides the framework, abstractions and language with which to implement the modular programs; Seddacco provides the language through which these programs are combined and controlled. The major divergence in these philosophies 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 a circumscribed form of metalinguistic abstraction – a principle supporting the rich LISP family of DSLs [3] – to allow application teams to customize much of the language's syntax to their domain.

5.3.2 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 think 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, promote readability, and reduce the burden with which an application programmer might explain iEpi's configuration to a domain expert. The various clauses can be found in Table 5.1.

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 given below. The syntax changes slightly depending on the context for practical reasons. For instance, it makes little sense to provide a schedule to a task that will be executed exactly once at the beginning of the program:

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

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

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

5.3.3 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 [55] and (in a circumscribed form) metalinguistic abstraction [3], 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. They are often used to capture parameters, e.g., `collect <variable word> data` would match `collect gps data` and `collect CO`$_2$ `data` and then pass `gps` and `CO`$_2$, respectively, to the declaring iEpi construct. Constructs using these elements get and validate their text values at instantiation. For example, `collect hamburger data` would be linked to a specific iEpi construct matching the given token pattern, but would be an invalid parameter and thus rejected when validated.

Number. Any number. Numbers can be qualified, i.e., they can be forced to match a string pattern (e.g., "1.3e-5") or be in a range (e.g., [0,1]), and are given a name for later reference. Number tokens are processed in a way similar to variable word tokens.

Operator. Commonly used non-alphanumeric mathematical, monetary, and programming symbols, e.g., <=, %, $. Operator tokens are processed in a way similar to variable word tokens.

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. String lists are processed in a way similar to variable word tokens.

Bracketed Item List. A comma delimited collection of character sequences, where each sequence is delimited by brackets, e.g.,`[item 1], [item 2], [item 3]`. These support within-clause parameter lists. Bracketed item lists are processed in a way similar to variable word tokens.

In accordance with the Chain-of-Responsibility pattern [32], 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`. Multiple token patterns can be used to provide access to a single iEpi construct. These patterns can be extremely similar, e.g., supporting multiple variable words to facilitate statements such as `collect atmospheric` $\texttt{CO}_2$ `data`, or entirely disjoint. The ability to provide multiple disjoint patterns to a single construct is useful in supporting legacy constructs or to allow groups with different domain jargon to employ the same constructs. For example, a location acquiring task might be invoked by a sensor-oriented group using `query [gps], [network], [agps] coordinates`, whereas a human-oriented group might use `estimate participant location`; both would invoke the same functionality, but the former would allow for fine-grained sensor control and a sensor-centric way of thinking about the problem, as opposed to a simplified, participant-centric approach. Employing this pattern helps satisfy our extensibility, concision, and expressiveness requirements. A simplified example of the matching process can be found in Figure 5.1.

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 that 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, domain-appropriate abstractions, and syntax requirements; the Parameter Clause is especially useful when specifying long parameters such as those given to an encryption algorithm, e.g., `use rsa encryption with [modulus 123456789012345678901234567890...], [exponent 4567890123...]`.

5.3.4 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, Schedule, 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) [32]. 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, tasks, and schedules to be parsed, which has been demonstrated to confer great benefits in the domain-specific language context [3]; schedule flexibility is limited due to the only recent introduction of a schedule-specific construct parser and restrictive schedule architecture with which it must interface.

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. In principle, it seems that all clauses as used in the current version of Seddacco could be uniquely identified by a handful of keyphrases (e.g., only if, with, then, and and/or/not), a list item separator (i.e., a comma), and a statement-ending symbol (e.g., a semicolon or a period).

An implementation using only these symbols would virtually eliminate programming symbols from the language's clauses if one discounts user-introduced tokens in token patterns. For ease of implementation and parsing, however, Condition, Parameter, and Schedule clauses are currently delimited by square brackets. These allow Seddacco's ANTLR grammar to quickly and unambiguously extract them from a statement. Without symbols, a token consumption scheme would be required for the Task and Schedule clauses to differentiate their elements; parsing errors or ambiguities would require a scheme more complex than that which is currently implemented to determine which Tasks and Schedules the user wished to invoke. This was not attempted due to implementation uncertainties, recent time constraints, and our longstanding use of a single schedule type. Mandatory bracketing, unfortunately, reduces readability and should ideally be removed in the future.

5.4 Meeting the Specification

In section 5.2, six requirements were specified that Seddacco must meet to interface with its target platform iEpi and three requirements that it must meet to be a language appropriate for the use case realized through the iEpi architecture.

5.4.1 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:

Table 5.1: 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. To visually and syntactically delineate schedules, their non-burst length components are surrounded by brackets.	[every 10 minutes] [every 5 minutes] for 30 seconds [continuously] [forever] [about 10 times every hour] for 5 minutes [when iepi starts] for 10 minutes
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$_n$

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, or parameters. They are automatically joined when processed by Seddacco's parser.

5.4.2 Domain Requirements

Through the various clauses, Seddacco meets the three previously advanced conditions 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 that were stated:

Reusability. Seddacco promotes reusability in the same way that Unix command line shells do – via 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.

Decoupling. 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, larger data manipulation 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 [62, 87], which literally makes it an interface to an underlying implementation.

Metalinguistic Abstraction. Seddacco's token patterns and custom parsers allow interdisciplinary teams to readily and modularly customize the language and its jargon to be readable, usable, and understood by all members.

5.4.3 Fundamental Use Case

Seddacco fulfils the needs of the fundamental use case on a variety of fronts. 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 and used by invoking their custom-made token patterns. 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, tweaking their operation via parameters and the Parameter Clause, and, to a limited degree, customizing the schedule on which they run via the Schedule 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. Larger and more complex data processing can be configured and managed via Chained Task Clauses.

5.5 Examples

The syntax of Seddacco is presented as it might be given in three scenarios: a canonical [37] 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.

5.5.1 Ad-hoc Localization

Consider a hypothetical set of statements for on-device localization. In the Seddacco code below, localization is performed and used to update the user's position on a map by attempting to collect a ten second sample of GPS and WiFi data every thirty seconds. When GPS data is available, it is collected and used unconditionally to update the map. When it is not available or is inaccurate – e.g., when the user is indoors – WiFi data is used to locate the user and update the map; this context-limiting step is performed by the condition clause `only if [gps signal unavailable within last 30 seconds] or not [gps signal accurate to <= 100 meters]`. When WiFi data is collected to generate position, it is processed using a localization service identified in the `localize via skyhook` statement accessed from the address given in the subsequent parameter. When data is collected, it is output for storage via the implicit task chain and used to update a map. The Java code needed to implement the constructs and token patterns are not provided for brevity.

```
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;
```

5.5.2 Simple Mass Data Collection

iEpi was originally conceived for mass data collection. The code block below depicts the Seddacco configuration used in the SHED3/SHED4 [77, 78] experiments, updated to match the language's current syntax. In the startup block, encryption and device anonymization are configured. Below this, several sensors are told to operate on synchronous schedules. Most sensor network hardware must carefully ration sensor usage as they are, relative to the limited power capacity of the devices and their efficient CPUs, very energy intensive to manipulate. With Android, the platform on which iEpi operates, the opposite is true: sensors generally consume insignificant amounts of energy compared to simply waking the power-hungry CPU for program execution. SHED3 and SHED4 attempted to mitigate this energy consumption by firing all sensors simultaneously. They also reduce CPU and memory contention by batching data storage, which can be seen in the `store volatile data` statement. Data was uploaded opportunistically when the device was connected to a power source and infrequently otherwise: this was an attempt to balance likelihood of upload with in-the-field power consumption. Surveys and system configuration updates took place periodically throughout the day.

```
# Startup tasks configure encryption and user anonymization.
before tasks start:
  use custom rsa encryption with [modulus 5555], [exponent 5555];
  anonymize user identifiers with [anonymization key ‘‘aabbccdd’’];
done;

# SHED3 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 about three times every day between 8:00am and 6:00pm.
issue ‘‘midday-survey.xml’’
  [about 1 time every 4 hours]
  only if [hour between 8 and 18];
```

5.5.3 Complex On-Device Processing

Many traditional sensor network applications collect and apply operations to data off-device, which is often due to limited processing and battery power. With modern smartphones, this limitation is largely gone, allowing more intensive processes to be performed on-device. One example of this is WiFi-based positioning. WiFi-based positioning is typically used for indoor localization tasks where GPS performs poorly or not at all. The task involves the conversion of WiFi data to a precise position on a map. As demonstrated by Petrenko *et al.* [77, 78], this process can be complex and may require data filtering, trilateration, and map manipulation; if these steps are not performed, position estimates may be erratic, unstable, and significantly less accurate. By performing these operations on-device, wireless communication costs could potentially be reduced, saving power for other purposes. It could also allow WiFi-based positioning to be performed without a data connection as all relevant information would be stored on-device. The code block below depicts a hypothetical Seddacco configuration for such a positioning task chain.

```
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;
```

5.6 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 [7, 68]. Bai's classification system [7] 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 [68] 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. Although they may exist given the size of the literature on these subjects, the authors are not aware of other behavioral sensing macroprogramming languages that provide this exact feature set.

From a language perspective, Seddacco's use of metalinguistic abstraction appears to set 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 other languages such as LISP [3] benefit from metalinguistic abstraction, I am unaware of any configuration language that uses that capacity to benefit non-technical end users. Seddacco's innovative means of language extension further raises the intriguing future 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 the easy specification and use of data processing pipelines to perform more complex operations. Tasks and other constructs are declared in a human-friendly fashion that allows for mid-declaration parameters, and the end user does not have to declare how these linkages are formed. 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.

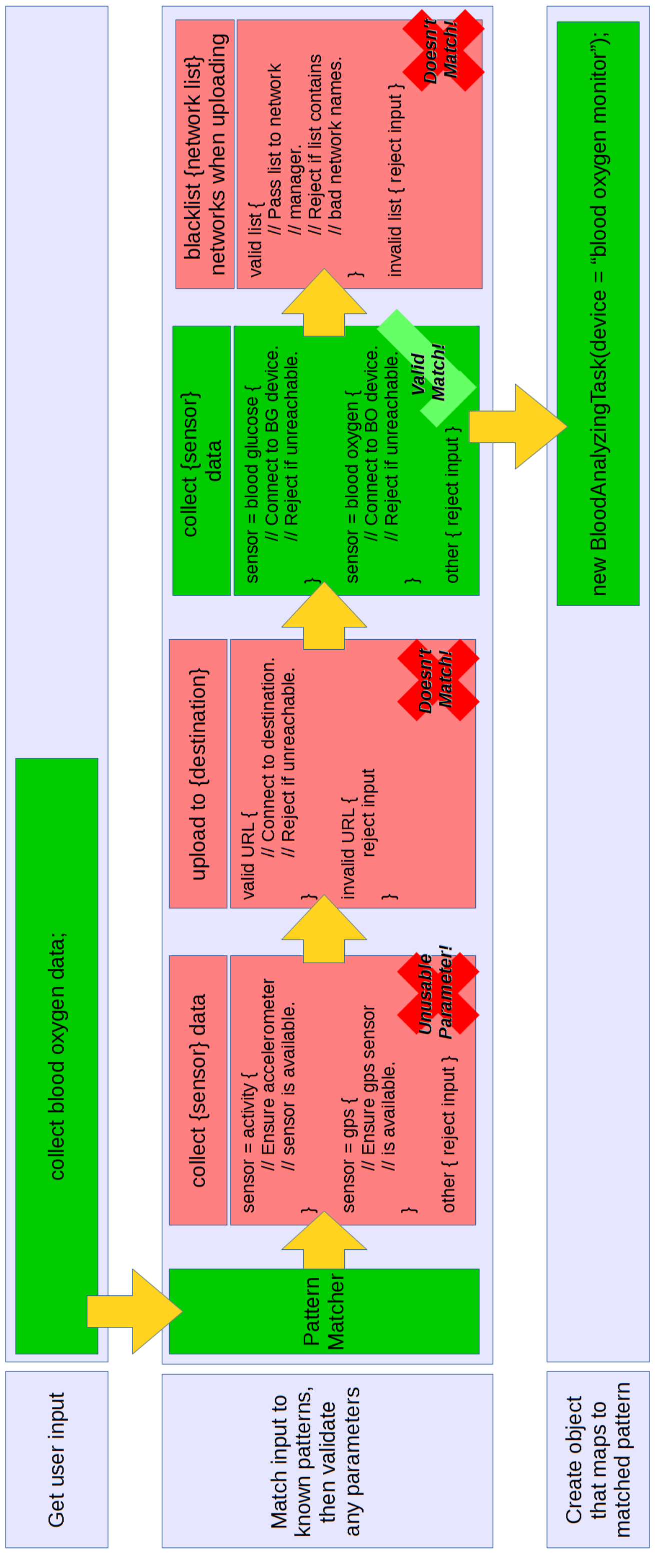

Figure 5.1: Creating a BloodAnalyzingTask instance from "`collect blood oxygen data`". Pattern matching process summarized using pseudocode.

Chapter 6

Discussion of Thesis Contributions

This thesis has made two major contributions: the data collection engine of iEpi, informally known as "the logger", and the configuration language, Seddacco, that powers and configures it. The work of this thesis has also significantly shaped the overall design of the entire iEpi system. While not a formal scientific contribution, it is notable that the designs, decisions, and work of one master's project has had such far-reaching effects.

6.1 Contributions of the Logger

iEpi's data collection engine makes several contributions to the field of data collection and to interdisciplinary research efforts:

Re-use of Pipeline Architecture. The system demonstrates that an architecture employed in operating system, compiler, and command line environments (among others) is useful and desirable in smartphone-based systems seeking to collect data on human behaviour.

Smartphone Viability as General-Purpose Research Devices. The literature and industry offer a growing number of examples wherein smartphones and cellphones are used as data collection devices. These efforts demonstrate that such devices can be used to collect data, but their use is limited to a narrow range of tasks, or they have not been deployed beyond their original environment. The logger, as thoroughly discussed in chapter 4, has been used in a variety of experimental settings, has facilitated topically disjoint human behaviour data collection endeavours, and has demonstrated its generality, extensibility, and flexibility through integration with new sensor types and external devices. The logger is also general enough to be used as a teaching tool: a class in our department has used the tool to conduct class-based experimental work.

Unifying Architecture. The logger's architecture is flexible enough to incorporate architectural concepts from the wireless sensor network and personal data collection literature without compromising its current structure. Because of this, the system makes it possible to test other researchers' prototype systems, e.g., a context-monitoring engine [45, 46], in actual deployments or in deployments not under the strict control of the original authors. This would allow for replication, falsification, and qualification of other scientists' results – a key goal in science. The architecure also supports interdisciplinary work. To date, the architecture has proved general enough to help study human mobility in care facilities, diabetes, influenza spread, tobacco-related behaviour and messaging, and population well-being.

Elevating the Standards of Software Development for Behavioural Data Collection Systems. The logger was developed from the start with testability and robustness in mind. The mechanisms that make this possible, i.e., modularity, decoupling, and test-driven development, are documented in the papers published on the logger. This sets a precedent for future designers of data collection systems: if a complex data collection system can validate its stability and correctness, and be evaluated by real-world usage, then there is little justification in using less rigorous design techniques to build similar systems in this domain.

Hybridization of Traditional and Technological Tools. The logger allows the best of both traditional and novel collection tools: researchers can use sensor hardware to collect quantitative data while still using traditional, more familiar tools to collect qualitative information. Each such domain of data can be used to reinforce conclusions or identify problems identified by the other, further bolstering their utility. Data fusion between these sources also allows for greater understanding of the user's context and, when exploited intelligently, a better understanding of phenomena under scrutiny. For example, a study of jogging's impact on emotional well-being could employ surveys to assess emotional state and GPS, accelerometer, and heart rate data to track the user's activity habits: data fusion could help the researcher identify if emotional benefits were most strongly derived from high-quality activity, high-quality or pleasant locations, or both. Importantly, the qualitative data can help interpret the associated sensor data.

Future Avenues for Research. The logger is stable enough to be used for future collection efforts. Proof of this lies in close to a dozen studies both at home and abroad that have been facilitated through this work. Because of the logger's proven stability and extensibility, the research community now has a platform for the collection of health behaviour data. As measures are gradually introduced to further improve the system's flexibility, the ease with which researchers can utilize the system should only increase.

Six Publications. This system has directly facilitated six papers, [38, 39, 49, 50, 77, 78]. It seems likely that it will soon facilitate more publications that exploit the data it has already collected.[1]

6.2 Contributions of Seddacco

Seddacco and the paper that describes it have also contributed to the fields of data collection, domain specific language design, and to interdisciplinary research:

[1]For example, a paper concerning Exflu has recently been submitted to PloS One. Data returning from other collaborations in Boston and Saskatoon also show early promise.

Metalinguistic, English-Like Syntax. To the extent of my knowledge, there is no other configuration language for data collection tools that has both an English-like appearance and allows its syntax to be partially altered. Many languages have syntax that read like English – some being so English-like that they can be used to write interactive fiction and themselves read like stories [71] – but do not appear to have the capacity for new, user-defined syntax. The confluence of such abilities is particularly important in the context of teams that are both interdisciplinary and working in particular domains. Through the use of token patterns, Seddacco allows developers to create sub-parsers to interpret their own custom syntax within the simple, encompassing clause framework that Seddacco provides. This also allows developers to place parameters wherever they deem most appropriate, which itself appears to be a rare language feature. Other languages featuring metalinguistic abstraction, but without English-like syntax, include Lisp [3] and Haskell [41].

Guidelines for DSL Design. The paper in which Seddacco appears details language features that should be present in systems collecting user data from smartphones. These features may act as a guide for future developers attempting to create DSLs in this realm.

Interdisciplinary Communication. Anecdotal evidence suggests that Seddacco supports communication with domain experts more easily than other programming languages. This may allow for better communication between domain experts and developers, and provide a common place for experts in different domains to communicate. Given modifications in chapter 7 that could improve the readability of its syntax, Seddacco could be made more friendly to these audiences.

6.2.1 Logger / Seddacco Synergy

Taken together, the logger and Seddacco provide an environment in which data-hungry researchers can specify their collection needs, have developers implement those needs as slot-in additions to a well-tested system, and then rapidly deploy their experiment through the use of a high-level configuration language.

6.3 The Big Picture: What Problems Were Solved?

This thesis began with a discussion of the state of behavioural data collection. It was noted that behavioural data collection has been limited by traditional tools and quality problems that plagued them. Custom tools were developed, but were often fragile, ethically problematic, or disrupted participant behaviour. These tools were also difficult to manage and distribute. Once deployed, it was not always clear how reliable they were, both in terms of stability and reported data quality. As the work of this thesis is yet another tool, it is natural to ask the following question: how does it solve the identified problems that other tools have largely failed to address?

The logger improves on traditional tools in several ways. By default, it provides mechanisms to use those tools directly on the user's device. It can launch external survey applications, diarying applications, and can even be augmented to incorporate new UI elements not easily placed in paper documents. The logger aims to be an objective data collection tool: unless explicitly instructed otherwise, it does not judge what data is useful and what is not, nor does it make interpretations of data. As the logger normally operates silently in the background of participants' phones, the data it collects appears less likely to be warped by social norms, embarrassment, and other cultural factors that are introduced when participants directly supply data to researchers or surveys. Assuming the participant carries his or her phone more consistently than surveys, travel diaries, and other researcher-supplied materials, the logger is less likely to suffer from human memory problems that introduce bias, holes, and distortion, and does not alter collection regimes if the user is busy or forgetful. The logger, being mounted on a phone with which the user is familiar and is likely to carry with them, is generally unobtrusive and its presence can likely be forgotten; in fact, to prevent the participant from forgetting that they are being monitored, the logger was explicitly designed with a startup reminder.

The logger also improves on specialized collection hardware such as pedometers. If these devices are equipped with certain types of radios, the logger can be extended to communicate with them. The logger incorporates many sensor types, allowing it to perform many of these simple, specialized features, such as location tracking, without additional modification. Thanks to its smartphone hardware base, it has the benefit of being non-obtrusive, sturdy, and powerful. The logger removes problems noted in the review of ethically ambiguous celltower data collection: the software must be explicitly installed on each user's device, it notifies them of its presence, it anonymizes and encrypts their data, and can be uninstalled by the user at any point. Additional privacy features – e.g., a filter in the implicit task chain that removes non-administrative data when enabled by the user in private moments – also exist in the system; new privacy features can also be added to the system when required with generally little effort.

The logger and Seddacco work together to resolve problems encountered by other smartphone tools. The logger is highly decomposable, modular, extensible, flexible, and robust. It is well tested, with nearly 900 automated tests ensuring its correctness and over 900 more assertions checked repeatedly at runtime to detect bugs not caught during formal testing procedures. It can be used in a variety of studies, and can be extended in ways not originally envisioned by its creators. Seddacco allows all of these features to be cleanly encapsulated in a high-level language, while still allowing fine-grained parameters to be passed to the underlying system for the purposes of optimization. In sum, these tools are reliable, can be used to improve collected data quality, and can be used in a wide variety of use cases.

Chapter 7
Future Work

There is a substantial amount of future work to be done with the iEpi system, the logger, and Seddacco if their continued use is desired. This work includes a variety of functional improvements to address blocking usability, efficiency, and reliability problems. For example, the logger's between-task data transfer mechanism enforces lockstep data transfer, which causes unnecessary bottlenecks and processing inefficiencies. The prospective work also includes numerous non-essential improvements that could increase the appeal and usability of iEpi. For example, an improved Seddacco syntax would make it easier for domain experts to interpret, and mobile tasks like those employed in [28] could make it easier for research teams to distribute specialized monitoring, query, or maintenance code in real time. Security flaws are also documented for completeness, as these are not obvious without an accurate, detailed understanding of the system's inner workings.

7.1 The Logger

7.1.1 Short-Term Improvements

The following improvements should be made at the first available opportunity to ensure the future stability, performance, flexibility, and usability of the logger.

Immediate Condition Updates. The logger has a limited condition system. Conditions were traditionally supported through the use of a cache. These caches have been awkward to use, buggy due to Android's unreliable SQLite implementation, and difficult to augment for new studies. A more efficient condition system such as [45, 46], or one that automatically redirects data to components that need it [75, 76] would improve the system's performance. A system of this type would allow an event triggering system to exist. At any point, constructs dependent on conditions can also simply check their boolean status and evaluate internal conditions as need be.

This new condition system has the potential to significantly reduce the overhead and bugs associated with the existing cache system, and to make context recognition and condition evaluation possible at the moment changes occur. It is important to note that most new constructs are being developed with this eventuality in mind. iEpi has been evolving to make this modification a reality: data labels are already in place in the current iEpi system, and some direct "task-to-condition" data connections exist for specific deployments.

Task Infrastructure Improvements. The logger's Task and Task Chain infrastructure is a usable pipeline system but, when compared with the system used in tools such as bash [29], it is suboptimal. For example, tasks require registration and deregistration when data is to begin and cease flowing. These operations are effectively invisible to programs in the bash environment. Tasks within a process must move in lockstep as data moves through them – a flaw noted only very late in the design process. Conditions only work for the first task in a task chain, despite their ability to be associated with any tasks. Much of these difficulties could be resolved by following the bash model more explicitly. Tasks in a task chain should further be given their own threads, as they are currently grouped such that they exchange data in lockstep.

Improved Energy Efficiency. The logger's dependence on WakeLocks [5] and Android's energy-heavy CPU usage make the logger less energy efficient than custom data logging hardware. A full energy usage profile would better inform future development and optimization. Additional energy efficiency techniques, such as adaptive sampling and actuation, should further reduce power consumption. Other authors also define techniques for energy costs and budgets to be specified through the use of credits [68] and/or marketplace approaches [61]. These latter approaches could be used to ration power on a device to ensure a more long-lived operation.

7.1.2 Desired Improvements

There are a number of improvements that could be made to the logger to improve its efficiency, ease of use, and ease of extensibility.

Remove the RecordDatabase Class. The central RecordDatabase class accepts all data from the central data receiver, i.e., the DataLogger class, for non-volatile storage. The database must periodically be flushed to a file. Volatile data must also be inserted in batches into the database because SQL insertions in Android seem slow and inefficient. Instead, the system could accumulate data in memory and directly write it to file in batches when enough data is acquired.

Make the Implicit Task Chain a Proper Task Chain. The implicit task chain is actually a series of non-task classes adapted to the system's data exchange interface. Despite the remainder of the system treating the implicit task chain as if it were a task chain, this should be refactored so that the logger is more logically consistent. This would also be desirable for deployments that have non-standard, all-encompassing data processing needs, as it would allow the chain to be modified as need be.

Schedule-Task Feedback. Schedules cannot currently receive feedback from conditions or tasks. This should be resolved, as it would allow for adaptive sampling. Alternatively, the suggested system to perform condition updates could be made accessible to schedules, which would fulfil a similar role.

Randomized Schedule. It would be highly useful to have a schedule akin to `do something X times every <time interval>`. This would make actions that need to be performed at random intervals, like issuing questionnaires, significantly simpler. The current approach exploits an aggressive duty cycle and a probability-based condition, which has proven to be error-prone and tedious during configuration.

Privacy Mode Refactoring. Privacy mode, colloquially referred to as "snooze mode", is implemented in a potentially inefficient manner.

Easier Interaction with External Tools. There are currently very few abstractions to allow the user to interact with Bluetooth-connected hardware. Abstractions to make this process easier, or at least more streamlined, would be highly desirable.

Networking Operators. Some languages support local-area primitives, e.g., [37], or network-wide operators, e.g., [59], to facilitate network communication and data acquisition. Simple support for these operators could be introduced to emulate or facilitate these features, which can be desirable when a population-level view of some phenomena is required.

7.1.3 Wish List

Injectable Agents. Fok [28] describes Agilla, a system in which agents can be injected into a sensor network and then allowed to run, migrate between systems, and interact with other agents. iEpi could benefit from their implementation. This is particularly true for tasks that need to be distributed mid-deployment, e.g., a critical update operation, or for creating entities that are naturally represented by dispersion, e.g., simulated viruses. Similar ideas can be found in Boulis' work [11], in which mobile control scripts are used to reconfigure devices and distribute new services mid-deployment.

iEpi Battery Calibrator. The logger is battery intensive. From initial investigation, however, this appears to be largely due to the way Android manages a device's CPU. Nevertheless, sensors and various operations cost energy. It would be wise to produce a tool that automatically ran the logger's functionality to assess its power impact on each type of device. This could be used in conjunction with, for example, a credit-based approach [66], wherein each action had an associated cost. More costly actions might only be fired above an energy threshold, or only infrequently below a certain energy threshold.

7.2 Seddacco

Seddacco has potential to become a more general-purpose programming language, but this will require significant work to achieve. Performing this work might create a natural language programming environment in which domain experts can more easily interact with each others' tools and exploit modern features, such as concurrency.

7.2.1 Short Term Improvements

Scheduling Syntax Improvement. To facilitate new scheduling algorithms for a recent prototyping effort, the Schedule Clause's syntax was changed to require a bracketed item before the burst length (i.e., `for <interval>`) declaration. This leaves Seddacco's syntax with an unnecessarily awkward scheduling clause. This can be resolved through a greedy token consumption scheme. Because the task and schedule clauses are not separated by a keyword, they must be acquired by ANTLR as a single unit. The longest match (without instantiation-time validation) of all known task token patterns should then be applied to the tokens of this larger entity to extract a task. The remaining tokens should then be treated as the Schedule Clause.

7.2.2 Desired Improvements

There are a variety of desired additions that could be made to Seddacco. Many of these would require syntax augmentation; those that do are provided with a suggested syntax and possible examples to help illustrate what is desired.

Parallel Task Block. Parallelization is increasingly important in computing applications. The logger currently supports data output from one task to multiple sources, but no syntax for this functionality is exposed in Seddacco. Used correctly, this would allow for efficient, parallelized processing of data that is produced.

Syntax Suggestion:

```
<task chain>
then simultaneously
  <task chains>
(done; | when done <task chain>)
```

Examples:

```
collect gps data
every 10 minutes for 30 seconds
then simultaneously
```

```
    update map icon;
    stream location to server;
  done;

  collect accelerometer data
  every 10 minutes for 30 seconds
  then simultaneously
    estimate activity type;
    estimate activity intensity;
  when done calculate calories burned;
```

Standalone Parallel Task Block Similar to a parallel task block embedded within a task chain, this block allows actions to be performed in parallel. The standalone version, however, would be used to synchronize several tasks on a single schedule.

Syntax Suggestion:

```
simultaneously <schedule>
  <task chains>
(done; | when done <task chain>)
```

Examples:

```
simultaneously every 10 minutes for 30 seconds
  collect wifi data;
  collect gps data;
done;

simultaneously every 10 minutes for 30 seconds
  collect wifi data;
  collect gps data;
when done estimate location;
```

Events. Events are useful when attempting to detect moments and context changes of interest. Rather than providing arbitrary schedules for tasks needing to respond to certain contexts, it would be useful to have a syntax that allowed tasks to be notified when other tasks identified the context in which they were interested. The following syntax is inspired by [45] and [46], as it mimics their functionality.

Syntax Suggestion:

```
when <condition expression> becomes <boolean>
```

```
    <parallel task chains>
  done;
```

Examples:

```
  when [outside] and ([raining] or [snowing]) becomes true
    issue ‘‘bad weather survey’’;
    sample accelerometer for 30 seconds;
  done;


  when [active] and [outside] becomes false
    show popup with [message ‘‘Quitter!!’’]
  done;
```

Code Definitions It is difficult to connect user-defined Seddacco statements to their underlying Java implementatiom. This could be made simple through Seddacco syntax; their implementation could also be made in the language itself, much like ANTLR allows Java to be written between { and }. Alternatively, ANTLR could be used by developers to directly specify token patterns in a well-known grammar and state how these token patterns are to be interpreted.

Syntax Suggestion:

```
  define <component type>
    named <component name>
    with token patterns <bracketed list>
    [with required parameters <bracketed list>]
    [with optional parameters <bracketed list>]
    <with implementation <implementation code> | linked to class <class name>>
  done;
```

Examples:

```
    define task
      named ‘‘Sampling Task’’
      with token patterns
        [sample word(sensor name)],       # e.g., sample gps
        [collect word(sensor name) data]  # e.g., collect wifi data
      with optional parameters
        [record cap of number(max data) data points]
      with java implementation
        // java code
        // code can reference variables ‘‘sensor name’’ and ‘‘max data’’
    done;
```

Task Chain Naming It would be highly useful to encapsulate task chains in their own task to make it easy to refer to them elsewhere in code.

Syntax Suggestion:

```
  Similar to code definitions.
```

Examples:

```
    define task
    named ‘‘Indoor Localization’’
    with token pattern
      [estimate indoor location]
    with task chain implementation
      collect wifi data
      then trilaterate
      then ...
    done;
```

Domain Expert Friendly Syntax Wide changes to Seddacco's syntax could make it easier for domain experts to use and read. Specifically, making the language case insensitive, replacing "then" with a comma or "<comma> and then", replacing the end-of-statement semicolon with a period, making text and numeric representations of numbers synonymous, and making the parser more intelligent, such that bracketed lists were not required in schedules, conditions, and parameters, would likely improve its readability.

Examples:

```
  Parse data in ‘‘input.csv’’,
    translate data to Excel format,
    and then run "aggregation" macro with column range ‘‘B3:F9’’.


  Collect GPS data about three times per day for 30 seconds
    only if outside and moving and not near WiFi network.
```

7.2.3 Wish List

Migration to Non-Smartphone Platform. Other languages and platforms exist that are used to perform pipelined data processing. A famous example of this is bash [29] used in Unix-like environments. Bash scripts, however, are infamous for becoming terse and cryptic if not carefully documented. Seddacco could be ported to these environments and used as a more user-friendly shell interface. As a variety of domain experts use bash scripts to process data in ways that are opaque to experts in other domains, a "Seddacco Shell" could further be employed as a human-friendly, readable bridge between these domains.

7.3 Known iEpi Security Vulnerabilities

While we have made every effort to ensure the security of the logger, Seddacco, and iEpi as a whole, known vulnerabilities exist.

Anonymization. User anonymization does not prevent inference attacks: if the experimenter knows the home of an individual, they can use location data to infer the participant mapped to a specific identifier. Succeptibility to these kinds of attacks can be reduced via techniques described in [12].

RSA ECB Weakness: Due to early system limitations, iEpi's RSA encryption uses an electronic codebook (ECB) scheme, which is vulnerable to certain kinds of attacks. This could be replaced by a better scheme.

Limited Encryption Scope. Survey responses are not encrypted (as they are generated outside of the logger, which performs encryption). Data stored in the researcher's database is not encrypted; rather, the data storage server can only be accessed by authenticated individuals. Data is uploaded to this server via a DMZ server, which itself does not host any data.

Chapter 8

Conclusion

This thesis has detailed the creation and use of an extensible, flexible, and well-tested data collection engine and an accompanying language to configure it. These systems form an integral part of the larger iEpi system and facilitate its data acquisition, allowing a variety of data to be collected from a growing number of sources with increasing contextual and temporal specificity. They have been used in a large number of deployments, growing and evolving to fit the needs of each new experiment. They are built for future expansion and possess a roadmap to guide future developers, students, and collaborators tasked with their improvement.

These contributions address problems of flexibility, reliability, and human error faced when collecting behavioural data. They work around the problems of traditional tools and their proposed replacements by providing objective, quantitative, and ever-present contextual data collection that can be tailored and extended to meet a study's needs. These contributions improve confidence in automated data collection, as they have been used in numerous field studies (chapter 4), operate on durable platforms that are often carried by their users, and compensate for human factors such as forgetfulness, bias and carelessness by complementing participant self-reporting with pervasive and/or contextually collected sensor data. Facilities for easy system configuration, future expansion, and validation are also provided, which we believe will increase the use of this thesis' contributions in the future.

It is the hope that the system encompassing the works of this thesis, iEpi, may one day form a discipline-standard data collection system. The author's contributions have helped bring this vision closer to reality. By expanding on and exploiting its generality, expressiveness, and extensibility, this work may one day join the collection of academic tools seeing widespread use and popularity. Given the interest and excitement expressed by collaborators, other institutions, and some members of industry, this is not an unreasonable expectation.

References

[1] David M. Aanensen, Derek M. Huntley, Edward J. Feil, Fada'a al Own, and Brian G. Spratt. EpiCollect: Linking Smartphones to Web Applications for Epidemiology, Ecology and Community Data Collection. In *PLoS ONE*, volume 4, page e6968, 2009.

[2] Tarek F. Abdelzaher, Brian M. Blum, Qing Cao, Y. Chen, D. Evans, J. George, Selvin George, Lin Gu, Tian He, Sudha Krishnamurthy, Liqian Luo, Sanghyuk Son, John A. Stankovic, Radu Stoleru, and Anthony D. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 582–589, Hachioji, Tokyo, Japan, 2004.

[3] Harold Abelson, Matthew Halfant, Jacob Katzenelson, and Gerald J. Sussman. The LISP experience. In *Annual Review of Computer Science*, volume 3, pages 167–195, 1988.

[4] Android. AlarmManager, May 2014. [Online]. Available: http://developer.android.com/reference/android/app/AlarmManager.html.

[5] Android. PowerManager.Wakelock, September 2014. [Online]. Available: http://developer.android.com/reference/android/os/PowerManager.WakeLock.html.

[6] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming Heterogeneous Sensor Networks Using Cosmos. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 159–172, Lisbon, Portugal, 2007.

[7] Lan S. Bai, Robert P. Dick, and Peter A. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 85–96, San Francisco, CA, 2009.

[8] Scott Bell, Wook Rak Jung, and Vishwa Krishnakumar. Wifi-based enhanced positioning systems: Accuracy through mapping, calibration, and classification. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*, pages 3–9, San Jose, CA, 2010.

[9] Urs Bischoff and Gerd Kortuem. A State-Based Programming Model and System for Wireless Sensor Networks. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops.*, pages 261–266, White Plains, NY, March 2007.

[10] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 690–699, Hachioji, Tokyo, Japan, 2004.

[11] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. In *Pervasive and Mobile Computing*, volume 3, pages 386–412, August 2007.

[12] A.J. Bernheim Brush, John Krumm, and James Scott. Exploring end user preferences for location obfuscation, location-based services, and the value of location. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, pages 95–104, Copenhagen, Denmark, 2010.

[13] Germaine M. Buck Lewis and Rajeshwari Sundaram. Exposome: time for transformative research. In *Statistics in Medicine*, volume 31, pages 2569–2575, 2012.

[14] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, pages 233–244, Chicago, IL, 2008.

[15] Ciro Cattuto, Wouter Van den Broeck, Alain Barrat, Vittoria Colizza, Jean-Franois Pinton, and Alessandro Vespignani. Dynamics of Person-to-Person Interactions from Distributed RFID Sensor Networks. In *PLoS ONE*, volume 5, page e11596, 2010.

[16] Pew Research Center. Mobile Technology Fact Sheet, March 2014. [Online]. Available: http://www.pewinternet.org/fact-sheets/mobile-technology-fact-sheet/.

[17] Tides Center. Open mHealth, March 2014. [Online]. Available: http://openmhealth.org.

[18] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, pages 175–188, Sydney, Australia, 2007.

[19] Jeffrey A. Claridge and Timothy C. Fabian. History and development of evidence-based medicine. In *World Journal of Surgery*, volume 29, pages 547–553, 2005.

[20] The AnyLogic Company. AnyLogic: Multimethod Simulation Software, August 2014. [Online]. Available: http://www.anylogic.com.

[21] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks. In *Proceedings of the International Workshop on Middleware for Sensor Networks*, pages 43–48, Melbourne, Australia, 2006.

[22] Coen De Roover, Christophe Scholliers, Theo Amerijckx, Wouter; D'Hondt, and Wolfgang De Meuter. CrimeSPOT: A language and runtime for developing active wireless sensor network applications. In *Science of Computer Programming*, volume 78, pages 1951–1970, 2013.

[23] Nathan Eagle and Alex (Sandy) Pentland. Reality mining: sensing complex social systems. In *Personal and Ubiquitous Computing*, volume 10, pages 255–268, 2006.

[24] Farjana Z. Eishita and Kevin G. Stanley. Iterative Design of an Augmented Reality Game Editor for School Children. In *Grace Hopper Conference*, Minneapolis, MN, October 2013.

[25] Farjana Z. Eishita, Kevin G. Stanley, and Regan Mandryk. Iterative Design of an Augmented Reality Game and Level-Editing Tool for Use in the Classroom. In *IEEE Games, Entertainment, and Media Conference*. Minneapolis, MN, October 2014. (Accepted).

[26] Kai Elgethun, Richard A. Fenske, Michael G. Yost, and Gary J. Palcisko. TimeLocation Analysis for Exposure Assessment Studies of Children Using a Novel Global Positioning System Instrument. In *Environmental Health Perspectives*, volume 111, pages 115–122, Seattle, WA, 2003.

[27] Esri. ArcGIS: Mapping and Analysis for Understanding Our World, August 2014. [Online]. Available: http://www.esri.com/software/arcgis.

[28] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 653–662, Columbus, OH, 2005.

[29] Free Software Foundation. GNU Bash, August 2014. [Online]. Available: http://www.gnu.org/software/bash/.

[30] Christian Frank and Kay Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pages 230–242, San Diego, CA, 2005.

[31] Christian Frank and Kay Römer. Solving generic role assignment exactly. In *20th International Parallel and Distributed Processing Symposium*, pages 25–29, Rhodes Island, Greece, 2006.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[33] David Garlan and Mary Sha. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[34] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, CA, 2003.

[35] Marta C. González, César A. Hidalgo, and Albert-László Barabási. Understanding individual human mobility patterns. In *Nature*, volume 453, pages 779–782, 2008.

[36] Samsung Group. S Health: Your personal fitness tracker, August 2014. [Online]. Available: http://content.samsung.com/ us/contents/aboutn/sHealthIntro.do.

[37] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming Wireless Sensor Networks Using Kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems*, pages 126–140, Marina del Rey, CA, 2005.

[38] Mohammad Hashemian, Dylan Knowles, Jonathan Calver, Weicheng Qian, Michael C. Bullock, Scott Bell, Regan L. Mandryk, Nathaniel Osgood, and Kevin G. Stanley. iEpi: An End to End Solution for Collecting, Conditioning and Utilizing Epidemiologically Relevant Data. In *Proceedings of the 2nd ACM International Workshop on Pervasive Wireless Healthcare*, pages 3–8, Hilton Head, SC, 2012.

[39] Mohammad S. Hashemian, Kevin G. Stanley, Dylan L. Knowles, Jonathan Calver, and Nathaniel D. Osgood. Human network data collection in the wild: The epidemiological utility of micro-contact and location data. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, pages 255–264, Miami, FL, 2012.

[40] Mohammad S. Hashemian, Kevin G. Stanley, and Nathaniel Osgood. Flunet: Automated tracking of contacts during flu season. In *Proceedings of the 8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, pages 348–353, Avignon, France, 2010.

[41] HaskellWiki. The Haskell Programming Language, August 2014. [Online]. Available: http://www.haskell.org.

[42] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. MacroLab: A Vector-based Macroprogramming Framework for Cyber-physical Systems. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, pages 225–238, Raleigh, NC, 2008.

[43] Apple Inc. iBeacon for developers, September 2014. [Online]. Available: https://developer.apple.com/ibeacon/.

[44] Bernadette Kamleitner, Stephan Dickert, Marjan Falahrastegar, and Hamed Haddadi. Information bazaar: A contextual evaluation. In *Proceedings of the 5th ACM Workshop on HotPlanet*, pages 57–62, Hong Kong, China, 2013.

[45] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, pages 267–280, Breckenridge, CO, 2008.

[46] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Youngki Lee, Souneil Park, and Junehwa Song. A Scalable and Energy-Efficient Context Monitoring Framework for Mobile Personal Sensor Networks. In *IEEE Transactions on Mobile Computing*, volume 9, pages 686–702, 2010.

[47] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley, 1999.

[48] Darko Kirovski, Nuria Oliver, Mike Sinclair, and Desney Tan. Health-OS: A Position Paper. In *Proceedings of the 1st ACM SIGMOBILE International Workshop on Systems and Networking Support for Healthcare and Assisted Living Environments*, pages 76–78, San Juan, Puerto Rico, 2007.

[49] Dylan L. Knowles, Kevin G. Stanley, and Nathaniel D. Osgood. A Field-Validated Architecture for the Collection of Health-Relevant Behavioural Data. In *IEEE International Conference on Healthcare Informatics*, Verona, Italy, 2014. (In Press).

[50] Dylan L. Knowles, Kevin G. Stanley, and Nathaniel D. Osgood. Seddacco: An Extensible Language in Support of Mass Collection of Health Behavior Data. In *ACM SIGKDD Workshop on Health Informatics*, New York, NY, 2014. (In Press).

[51] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 200–210, San Diego, CA, 2007.

[52] Kyunghan Lee, Seongik Hong, Seong Joon Kim, Injong Rhee, and Song Chong. SLAW: A New Mobility Model for Human Walks. In *INFOCOM*, pages 855–863, Rio de Janeiro, Brazil, April 2009.

[53] Shuoqi Li, Sang H. Son, and John A. Stankovic. Event detection services using data service middleware in distributed sensor networks. In *Proceedings of the 2nd International Conference on Information Processing in Sensor Networks*, pages 502–517, Palo Alto, CA, 2003.

[54] Jong Hyun Lim, Andong Zhan, Evan Goldschmidt, JeongGil Ko, Marcus Chang, and Andreas Terzis. HealthOS: A Platform for Pervasive Health Applications. In *Proceedings of the Second ACM Workshop on Mobile Systems, Applications, and Services for HealthCare*, pages 4:1–4:6, Toronto, Canada, 2012.

[55] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Pearson Education, 2000.

[56] Imperial College London. EpiCollect.net, August 2014. [Online]. Available: http://www.epicollect.net.

[57] Hong Lu, Jun Yang, Zhigang Liu, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 71–84, Zurich, Switzerland, 2010.

[58] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks. In *ACM Transactions on Embedded Computing Systems*, volume 5, pages 543–576, New York, NY, 2006.

[59] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, number SI, pages 131–146, Boston, MA, 2002.

[60] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. In *ACM Transactions on Database Systems*, volume 30, pages 122–173, New York, NY, 2005.

[61] Geoff Mainland, David C Parkes, and Matt Welsh. Market-Based Programming Paradigms for Sensor Networks, 2004. [Online] Available: http://language6.com/m/market-based-programming-paradigms-for-sensor-networks-w4852.html.

[62] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. In *ACM Computing Surveys*, volume 37, pages 316–344, New York, NY, 2005.

[63] J. Scott Miller, Peter A. Dinda, and Robert P. Dick. Evaluating a BASIC approach to sensor network node programming. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, Berkeley, California, 2009.

[64] Chulhong Min, Chungkuk Yoo, Youngki Lee, and Junehwa Song. Healthopia: Towards your well-being in everyday life. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, pages 108:1–108:5, Barcelona, Spain, 2011.

[65] Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *Proceedings of the Second IEEE International Conference on Distributed Computing in Sensor Systems*, pages 150–168, San Francisco, CA, 2006.

[66] Luca Mottola and Gian Pietro Picco. Programming Wireless Sensor Networks with Logical Neighborhoods. In *Proceedings of the First International Conference on Integrated Internet Ad Hoc and Sensor Networks*, Nice, France, 2006.

[67] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods: A road tunnel use case. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, pages 393–394, Sydney, Australia, 2007.

[68] Luca Mottola and Gian Pietro Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. In *ACM Computing Surveys*, volume 43, pages 19:1–19:51, New York, NY, 2011.

[69] Rene Mueller, Gustavo Alonso, and Donald Kossmann. SwissQM: Next Generation Data Processing in Sensor Networks. In *Third Biennial Conference on Innovative Data Systems Research*, pages 1–9, Asilomar, CA, 2007.

[70] Yunyoung Nam, Seungmin Rho, and Chulung Lee. Physical Activity Recognition Using Multiple Sensors Embedded in a Wearable Device. In *ACM Transactions on Embedded Computing Systems*, volume 12, pages 26:1–26:14, New York, NY, 2013.

[71] Graham Nelson. *Natural Language, Semantic Analysis and Interactive Fiction.* 2005. [Online]. Available: http://www.inform7.com/learn/documents/WhitePaper.pdf.

[72] Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment Macroprogramming System. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 489–498, Cambridge, MA, 2007.

[73] Michael J. Ocean, Azer Bestavros, and Assaf J. Kfoury. snbench: Programming and virtualization framework for distributed multitasking sensor networks. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 89–99, Ottawa, Canada, 2006.

[74] Nathaniel D. Osgood, Roland F. Dyck, and Winfried K. Grassmann. The Inter- and Intragenerational Impact of Gestational Diabetes on the Epidemic of Type 2 Diabetes. In *American Journal of Public Health*, volume 101, pages 173–179. American Public Health Association, 2011.

[75] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. A Compilation Framework for Macroprogramming Networked Sensors. In *Distributed Computing in Sensor Systems*, volume 4549, pages 189–204, 2007.

[76] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 255–260, White Plains, New York, 2007.

[77] Anastasia Petrenko, Scott Bell, Kevin Stanley, Winchel Qian, Anton Sizo, and Dylan Knowles. Human Spatial Behavior, Sensor Informatics, and Disaggregate Data. In *Spatial Information Theory*, volume 8116, pages 224–242, 2013.

[78] Anastasia Petrenko, Anton Sizo, Winchel Qian, Dylan Knowles, Amin Tavassolian, Kevin Stanley, and Scott Bell. Exploring mobility indoors: an application of sensor-based and gis systems. In *Transactions in GIS*, volume 18, pages 351–369, 2014.

[79] Philip M. Podsakoff, Scott B. MacKenzie, Jeong-Yeon Lee, and Nathan P. Podsakoff. Common method biases in behavioral research: a critical review of the literature and recommended remedies. In *The Journal of applied psychology*, volume 88, pages 879–903, 2003.

[80] R Project. The R Project for Statistical Computing, August 2014. [Online]. Available: http://www.r-project.org/.

[81] Weicheng Qian, Kevin G. Stanley, and Nathaniel D. Osgood. The Impact of Spatial Resolution and Representation on Human Mobility Predictability. In *Web and Wireless Geographical Information Systems*, volume 7820, pages 25–40, 2013.

[82] Ipsos Reid. Close to Half of Canadians Now Own a Smartphone, March 2014. [Online]. Available: http://ipsos-na.com/news-polls/pressrelease.aspx?id=6005.

[83] Marcel Salathé, Maria Kazandjieva, Jung Woo Lee, Philip Levis, Marcus W. Feldman, and James H. Jones. A high-resolution human contact network for infectious disease transmission. In *Proceedings of the National Academy of Sciences*, pages 22020–22025, 2010.

[84] Noam Shoval and Michal Isaacson. Application of Tracking Technologies to the Study of Pedestrian Spatial Behavior. In *The Professional Geographer*, volume 58, pages 172–183, 2006.

[85] Queen's University Software Technology Laboratory. The TXL Programming Language.

[86] Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of predictability in human mobility. In *Science*, volume 327, pages 1018–1021, 2010.

[87] Diomidis Spinellis. Notable Design Patterns for Domain-specific Languages. In *Journal of Systems and Software*, volume 56, pages 91–99, New York, NY, 2001.

[88] Kevin Stanley, Farjana Eishita, Eva Anderson, and Regan Mandryk. Gemini Redux: Understanding Player Perception of Accumulated Context. In *IEEE Games, Entertainment, and Media Conference*, Oct 2014. (Accepted).

[89] Kevin G. Stanley, Ian J. Livingston, Alan Bandurka, Mohammad Hashemian, and Regan L. Mandryk. Gemini: A Pervasive Accumulated Context Exergame. In *Entertainment Computing*, volume 6972 of *Lecture Notes in Computer Science*, pages 65–76, 2011.

[90] Kevin G. Stanley and Nathaniel D. Osgood. The Potential of Sensor-Based Monitoring as a Tool for Health Care, Health Promotion, and Research. In *Annals of Family Medicine*, volume 9, 2011.

[91] Kevin G. Stanley, David Pinelle, Alan Bandurka, David McDine, and Regan L. Mandryk. Integrating cumulative context into computer games. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 248–251, Toronto, Canada, 2008.

[92] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. In *ACM Transactions on Sensor Networks*, volume 4, pages 1–29, New York, NY, 2008.

[93] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. In *Communications of the ACM*, volume 47, pages 34–40, New York, NY, 2004.

[94] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Education, Inc., 3rd edition, 2008.

[95] K. Terfloth, G. Wittenburg, and J. Schiller. FACTS – A Rule-based Middleware Architecture for Wireless Sensor Networks. In *First International Conference on Communication System Software and Middleware*, pages 1–8, Delhi, India, 2006.

[96] Yi-Hsuan Tu, Yen-Chiu Li, Ting-Chou Chien, and P.H. Chou. EcoCast: Interactive, object-oriented macroprogramming for networks of ultra-compact wireless sensor nodes. In *10th International Conference on Information Processing in Sensor Networks*, pages 366–377, Chicago, IL, 2011.

[97] Dashun Wang, Dino Pedreschi, Chaoming Song, Fosca Giannotti, and Albert-László Barabási. Human Mobility, Social Ties, and Link Prediction. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1100–1108, San Diego, CA, 2011.

[98] Matt Welsh and Geoff Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, San Francisco, CA, 2004.

[99] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 99–110, Boston, MA, 2004.

[100] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proceedings of the Third European Conference on Wireless Sensor Networks*, pages 5–20, Zurich, Switzerland, 2006.

[101] Christopher P. Wild. The exposome: from concept to utility. In *International Journal of Epidemiology*, volume 41, pages 24–32, 2012.

[102] David Wile. Lessons Learned from Real DSL Experiments. In *Science of Computer Programming*, volume 51, pages 265–290, 2004.

[103] Kue T. Young. *Population Health: Concepts and Methods.* Oxford University Press, 2nd edition, 2005. (See chapter 1, page 5).

[104] Yihong Yuan and Martin Raubal. Extracting Dynamic Urban Mobility Patterns from Mobile Phone Data. In *Geographic Information Science*, volume 7478, pages 354–367, 2012.