ABSTRACT

SOLLENBERGER, DEREK J. Engineering Personal and Social Affective Applications. (Under the direction of Dr. Munindar P. Singh.)

Affective (i.e., emotional) applications are becoming increasingly mainstream in entertainment and education. Yet, current techniques for building such applications are limited, and the maintenance and use of affect is in essence handcrafted in each application. The Koko architecture describes middleware that takes away the burden of incorporating affect into applications, thereby enabling developers to concentrate on the functional and creative aspects of their applications.

However, Koko can be effective only if the models it needs to function are suitably constructed. To this end, we have developed Kokomo, a methodology for creating affective applications. Specifically, it incorporates expressive communicative acts, and uses them to guide the design of an affective application. We empirically evaluate Kokomo’s utility through a developer study. The results are positive and demonstrate that the developers who used Kokomo, in conjunction with Koko, were able to develop an affective application in less time, with fewer lines of code, and with a reduced perception of difficulty than developers who worked without Kokomo.

To affective applications, Koko makes two contributions: improving developer productivity by creating a reusable and extensible environment; and yielding an enhanced user experience by enabling independently developed applications to collaborate and provide a more coherent user experience than currently possible. To agent-oriented software engineering, Kokomo contributes a methodology that incorporates expressives, which are largely ignored in conventional approaches, thus expanding the scope of AOSE to affective applications.

Further, we describe booST and hooPS, two applications built using Koko, which are targeted toward the youth demographic. These mobile, social applications are designed with the goals of promoting health and well-being by engendering and strengthening social relationships.
among youth. What makes booST and hooPS stand out is that they exhibit sensitivity to their user’s affective state and support the user in exchanging that state with other users. In this manner, these applications demonstrate how representing and reasoning about affect can facilitate deeper relationships that promote desirable behaviors and attitudes.
DEDICATION

This work is dedicate to my incredible wife, Brittany, who has stood by me through it all, and to my mom and dad who raised me to be the man that I am today.
BIOGRAPHY

Derek James Sollenberger was born in Chambersburg, Pennsylvania, to parents, James and Debra. He attended Cumberland Valley Christian School in his hometown, where he graduated in 2002. He then attended Shippensburg University for his undergraduate education. While there he majored in Computer Science with a minor in Mathematics. His decision to select a major in Computer Science was largely effected by his brother’s choice of the same career field as he himself had not prior experience in that field. During his undergraduate years, Derek was able to gain significant experience working with the Defense Information Systems Agency as a co-op during his junior and senior years. In December of 2005, Derek received his bachelor’s degree from Shippensburg University. The same month, he headed south, relocating in Cary, NC as he began graduate school at NC State University.

As a graduate student Derek joined the Multiagent Systems and Service-Oriented Computing Laboratory, led by his advisor, Dr. Munindar P. Singh. Under Dr. Singh’s direction Derek began studying the field of affective (emotionally-aware) computing. This led him to his interest in expanding the domain of emotionally-aware applications from the laboratory into mobile applications and existing software architectures. While a graduate student Derek, held various positions at technology companies. From 2006 through 2007 he worked as a software engineer for Motricity. In the Fall of 2007 he was employed by NC State as an instructor for their undergraduate computer networking course. Finally, in 2008 Derek spent the summer as an software engineering intern for Google. Since that time he has continued to work part-time with Google until transitioning to a full-time role in 2010.

Derek married his high school sweetheart, Brittany, in May 2004. The two of them share many common interests, including travel, which has resulted in visiting seven different Caribbean countries in their first six years of marriage. Finally, the two of them enjoy being actively involved with their local church family in Cary, NC and are happily awaiting of their first child in the Fall of 2011.
ACKNOWLEDGEMENTS

I would like to first thank the members of my committee, James Lester, Michael Young and Adriana de Souza e Silva for their guidance and support. Further, I am thankful for the opportunity to work with my advisor, Munindar P. Singh, who has given direction, support, and insight throughout my graduate studies and research.

I am grateful to Carol Wellington and Thomas Briggs who guided me through my initial years as a computer scientist at Shippensburg University. Your teaching and mentoring led me to pursue graduate studies and without your help I would not be where I am today.

I would like to express my sincere appreciation to my brother, Justin, and the rest of my family and friends who have, at some time or another, listened to me explain some aspect of my research. Furthermore, it is with the loving support, advice and encouragement provided by my parents, James and Debra Sollenberger that has enabled me to pursue my interests and flourish in whatever makes me happy. Finally, I would like to express my gratitude for the unconditional understanding, patience, and emotional support of my best friend and wife, Brittany.
# TABLE OF CONTENTS

List of Tables ............................................................................... vi

List of Figures ............................................................................. vii

Chapter 1 Introduction ................................................................. 1
  1.1 Motivation ............................................................................. 3
  1.2 Koko Architecture ............................................................... 5
  1.3 Kokomo Development Methodology ........................................ 8
  1.4 Summary of Contributions ................................................... 9
  1.5 Organization ........................................................................ 10

Chapter 2 Background .................................................................. 12
  2.1 Appraisal Theory .................................................................. 12
  2.2 Communicative Acts ............................................................. 16
  2.3 Evolution of Affective Gaming ................................................ 17
  2.4 Brief History of Mobile Social Applications .............................. 20

Chapter 3 Architecture .................................................................. 23
  3.1 Main Components and their Usage .......................................... 24
  3.2 Formal Description of Koko .................................................. 31
  3.3 Architecture Evaluation ....................................................... 44
  3.4 Deployment Approaches ...................................................... 48
  3.5 Design Rationale for Koko .................................................... 52

Chapter 4 Methodology ............................................................... 55
  4.1 Description of Kokomo ........................................................ 57
  4.2 Example Usage ..................................................................... 61
  4.3 Developer Study .................................................................... 65
  4.4 Empirical Evaluation ........................................................... 73

Chapter 5 Socio-Affective Applications ........................................ 81
  5.1 booST: Promoting Health ..................................................... 81
  5.2 hooPS: Strengthening Friendships ......................................... 83

Chapter 6 Discussion and Literature ........................................... 85
  6.1 Future Directions ............................................................... 91

References .................................................................................. 93

Appendix .................................................................................... 97
  Appendix A Developer Surveys .................................................. 98
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Types of mobile social applications</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Three states of CARE models</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Kokomo: a methodology for creating affective applications</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Subset of booST events</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Components of a Problem</td>
<td>69</td>
</tr>
<tr>
<td>4.4</td>
<td>Difficulty ranking of project components</td>
<td>77</td>
</tr>
<tr>
<td>A.1</td>
<td>Aspects of the Assignment</td>
<td>100</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Appraisal Theory [22] .................................................. 13
Figure 2.2 Monolithic programming model .................................. 17
Figure 2.3 Component-based programming model ...................... 18
Figure 2.4 Affective programming model .................................. 20
Figure 2.5 Koko programming model ........................................ 21
Figure 3.1 Koko architectural overview ..................................... 24
Figure 3.2 Main Koko entities .................................................... 26
Figure 3.3 Event ontology example .......................................... 31
Figure 3.4 Closed system deployment ....................................... 49
Figure 3.5 Global system deployment ....................................... 50
Figure 3.6 Hybrid system deployment with performance proxy .... 51
Figure 3.7 Composition of the booST and hooPS socio-affective applications. .... 53
Figure 4.1 Expressive message hierarchy .................................... 59
Figure 4.2 booST Screenshots ................................................... 62
Figure 4.3 Flow Channel .......................................................... 66
Figure 4.4 Application Components ......................................... 67
Figure 4.5 The application’s user interface ................................ 70
Figure 4.6 Software metrics ....................................................... 76
Figure 4.7 Development time metrics ....................................... 78
Figure 4.8 Subjective Developer Distributions ............................. 79
Figure 5.1 booST screenshots. (1) The home screen shows the user’s current status as well as a list of friends who also subscribe to booST. (2) The challenge screen where a user describes the conditions of the challenge he throws at a friend. (3) A screen showing a challenge that has been accepted, but has not yet been completed. .................................................. 82
Figure 5.2 hooPS screenshots from a student’s implementation. (1) The startup screen that displays the currently available basketball games. (2) The game screen that displays the status of the game as well as the messages exchanged between friends. (3) The message recommendation interface that provides suggestions based on both the sender and receiver’s emotional state. .................................................. 84
Chapter 1

Introduction

Commercial software developers spend significant amounts of time and effort tailoring their products to meet their user’s needs. However, in most cases developers do not know exactly who their end-users are. This separation between developers and users makes it nearly impossible for developers to craft a user experience that is ideal for all possible users. The field of affective computing helps reduce that separation and enables developers to get one step closer to providing the ideal user experience. Further, it is our goal to make that step possible by providing the tools and insights needed to bridge the gap between the innovations of affective computing and traditional software developers.

The term affect is used in psychology to refer to feelings or emotions. The term is also used more narrowly to describe an expressed or observed emotional response of a human to some relevant event. Expressed responses can be most easily demonstrated in a 3D virtual environment where virtual characters use gestures and dialog to emulate human emotional responses [22]. In contrast, we concentrate on the observation aspects of affective computing. In particular, we focus on enabling software developers to use affective computing techniques in order to create a superior user experience.

Recognizing and reasoning about affect enables developers to create systems that demonstrate superior intelligence [21], enhanced user interfaces [4], and more effective learning envi-
environments [14, 29]. However, games are among some of the most natural applications of affect. We refer to the genre of games that seek to incorporate their players’ affective state into the gameplay as affective or affect-aware games. Although many current affective games and applications are leading the way in their respective research areas, they all suffer from a few limitations. First, existing affective game developers are narrowly focused on ensuring the correctness of their affective model and rightfully so, since the correctness of the affective model is a necessary foundation for a suitably powerful game. This effort has resulted in games and applications that are built around the affect model, requiring that the game developer be an expert in the affective domain.

Further, there is no interoperability or continuity between affective games. For instance, most modern affect-aware games and their underlying affect model implementations are based on the discipline of appraisal theory [46]. A fundamental concept of appraisal theory is that the environment of an agent is essential for determining the agent’s affective state. As such, appraisal theory yields models of affect that are tied to a particular domain with a specific context. Therefore, each new problem domain requires a new affect model instance. A current and common practice has been to copy and edit a previous game (and, occasionally, to build from scratch) to meet the specifications of a new domain.

Copy and edit software engineering is clearly not a best practice, especially in light of modern advances in software engineering. Although the above limitations might not be prohibitive for constructing affective games as proofs-of-concept, these limitations render current techniques inadequate for developing production applications.

Our ongoing research program seeks to reduce the burden of incorporating affect into games. Our approach is twofold, in that we provide both tools and guidance to developers. The tools are packaged in a service-oriented middleware, called Koko, which observes and predicts the user’s affective state [47, 49]. The guidance comes by way of our methodology, called Kokomo, which steps the user through a process for configuring Koko and incorporating it into an application [50]. This document is broken into three logical units. The first unit covers the
Koko architecture. The second unit is centered on the Kokomo developer methodology that uses Koko to create affective applications. The final unit provides a look into the capabilities and utility of socio-affective applications. The remainder of this chapter briefly introduces these units, which will then be expounded upon in detail in their own chapters. We begin an brief overview of our motivation for supporting socio-affective applications since these motivations play a pivotal part in the design decisions for both the architecture and methodology.

\section{Motivation}

We started with this simple challenge: \textit{How can we connect to users, not just at the level of producing and consuming media, but also at a level which engages them both socially and personally?} This document describes the tools and techniques, in the form of Koko and Kokomo, we have developed to address precisely that challenge. However, before we elaborate on those items let us begin by briefly reviewing the top-level requirements resulting from this challenge that have motivated our research.

Our first requirement was to make sure users would be \textit{embedded} in the physical world, which we take to include the social world of people: friends, parents, and others. Moreover, it has been shown that integrating a user’s digital experience with that of the physical world can be used to promote learning and positive behavior \cite{4}.

Further, we took the position that everyone, and especially youth, would like to be known at a personal level. The feeling of being known as an individual and wanted as a friend yields a significant boost to one’s self esteem. Therefore, our second requirement was to promote \textit{personal} interactions.

A lot of the interactions mediated by today’s social networks tend to be of a superficial nature, for example, in terms of quick comments by users on pictures posted by one another. If we could strengthen and deepen such interactions in emotional terms, we would be able to use the connections among youth to more easily promote desirable behaviors and attitudes, for example, toward health and education. Therefore, our third requirement was to promote
emotional interactions.

Motivated by the above requirements we enable the creation of a class of mobile applications, which we call socio-affective applications. The goal of these applications is to enhance digital interactions by incorporating some of the key elements that make human-to-human interactions real. In particular, applications of this type meet our above requirements in the following manner.

**Embedded.** We focus our approach on mobility partly because today’s technologies are sufficiently advanced as to support high connectivity and partly because such emerging technologies are indeed the preferred way of communication for today’s youth. Further, mobile devices help us satisfy the requirement of being embedded. Additionally, emerging mobile pervasive technologies have enabled applications to augment the physical world. The resulting applications range from tracking your friend’s location (http://www.loopt.com) to creating games that use the physical world as an element of the gameplay [10]. Further, integrating the physical and virtual worlds has been shown to facilitate more compelling interactions than when considered independently [3].

**Personal.** We satisfied the requirement of being personal by focusing on applications where the communication is directed to one or a small number of people within a group. In this manner, we avoid the impersonal nature of many traditional digital media applications, especially traditional social networking.

**Emotional.** Finally, through Koko and Kokomo, we incorporate support for predicting user emotions, which is crucial to providing a realistic and customized user experience by having the application adapt itself to the user’s current emotional state. Emotions provide a new avenue for expression in and can help create applications that today’s youth would find engaging. Further, in social settings, users may expose their emotional state to their friends, thus conveying their personality and feelings to others.
1.2 Koko Architecture

Koko is a multiagent and service-oriented architecture, that enables the prediction of a player’s affective state. Koko is not another model of emotions but a (logically described) architecture within which existing (and future) models of affect can operate. Koko provides the means for both game developers and affect model designers to construct their respective software independently, while giving them access to new features. In doing so, Koko expands the state of the art by enabling multiagent affective interactions in ways that have previously been impossible. Additionally, the primary focus of the existing research has been modeling the affective state of a single agent. As a service-oriented architecture Koko builds on that foundation and goes further by incorporating affective computing with multiagent systems (MAS). As a multiagent system, Koko acts as the conduit for affective communication between agents, thereby enabling the development of a new class of applications that are both social and affective (i.e., socio-affective).

This section introduces two key aspects of Koko, namely its features and its ability to be coupled with existing application architectures.

1.2.1 Architectural Features

One of our primary motivations is to identify the common components in affective games and then to absorb those components into Koko. Two primary components of affective games are the affect model and physical sensors. The affect model is the part of the affective game that is responsible for computing an approximation of the player’s current affective state. Additionally, in order to more accurately predict the user’s affective state, many affective applications use physical sensors to provide additional information about both the user and the user’s environment [38]. The number and variety of sensors continues to increase and they are now available via a variety of public services (for example, weather and time services) and personal devices (for example, heart rate monitors and galvanic skin response units).

The abstraction of the above-mentioned components follows a pattern seen in programming
models for game development. Koko facilitates a new programming model for constructing affective applications. We elaborate on the historical programming models as well as the new Koko-based programming model in Section 2.3.

When compared to existing approaches for constructing affect-aware games, the Koko architecture provides following key benefits.

**Developer Productivity.** Koko separates the responsibility of developing the game from that of creating and maintaining the affect model by treating the game logic and the affect model as separate entities. By creating this separation, we can in many cases completely shield each entity from the other and provide standardized interfaces between them.

Additionally, Koko avoids code duplication by identifying and separating modules for accessing affect models and various sensors, and then absorbs those modules into the specified middleware. For example, by extracting the interfaces for physical sensors into the middleware, Koko enables each sensor to be leveraged through a common interface. Further, since the sensors are independent of the affect model and game logic, a sensor that is used by one model or game can be used by any other model or game without the need for additional code.

**User Experience.** Abstracting affect models into Koko serendipitously serves another important purpose. It enables an enhanced user experience by providing data to both the game and its affect model that was previously unattainable, resulting in richer game environments and more comprehensive models.

With current techniques it is simply not possible for separate games to share affective information for their common users. This is because each game is modeled, implemented, and deployed independently and thereby remains unaware of other games employed by its user. By contrast, Koko-based games can share common affective data through Koko. This reduces each application’s overhead of initializing the user’s affective state for each session, as well as provides insight into a user’s affective state that would normally fall outside of the game’s scope.

Such *cross-application sharing* of a user’s affective information potentially improves the
value of each game. As a use case, consider a student using both an educational and an entertainment game. The educational game can proceed with easier or harder problems depending on the user’s affective state even if the user’s state were to be changed by participation in some, otherwise unrelated game.

**Affective Social Applications.** In addition to enabling cross-application sharing of affective data, the Koko architecture enables cross-user sharing of affective data. The concept of cross-user sharing fits naturally into the realm of social and multiplayer games. Through Koko, an authorized user may view the affective state of other members in his or her social circle or multiplayer party. Further, that information can be used to better model the inquiring user’s affective state.

Further, Koko enables the development of affective social applications by introducing the notion of expressive communicative acts into agent-oriented software engineering. Using the multiagent environment provided by Koko, agents are able to communicate affective information through the exchange of expressive messages. The communication of affective information is naturally represented as an expressive communicative act [42]. However, expressive acts are a novelty in both agent-oriented software engineering and virtual agent systems, which have traditionally focused on the assertive, directive, and commissive communicative acts (e.g., the FIPA inform command).

### 1.2.2 Architectural Connectors

The Koko architecture as a stand-alone unit is not sufficient to build most complex applications. For example, 3D games rely on advanced game engines to create their applications. Even smaller scale games often leverage some third-party library to control certain gameplay functions or other intelligent behavior. To ensure that Koko can be leveraged in environments where it must work in conjunction with other modules, we have designed mechanism to enable interoperation between Koko and the other module. Another way of thinking of this is that Koko provides the connectors to enable it to be loosely coupled to the third-party architecture. The primary reason
for designing such connectors becomes more apparent when we discuss the social and multiplayer aspects of Koko, although such an approach is consistent even with existing techniques for multiplayer access to a central game server.

The interfaces that enable the coupling between Koko and other modules are naturally broken into two distinct categories: configuration and runtime. The configuration interfaces enable affective game developers to notify Koko of their game’s intent to use the middleware and also agree to the format of the runtime communication. The runtime interfaces are then used to send information needed by the affect model and to retrieve the user’s affective state or an approximation thereof. Both the configuration and runtime interfaces are formally defined in the mathematical Z notation [51]. The Z notation is an established way of describing software components and architectures and has been successfully applied in agent software as well [1].

We have carefully left the architecture open in that we declaratively specify the interfaces but do not restrict how the components implement those interfaces. This open architecture enables Koko to be deployed in a variety of configurations. In Section 3.4, we elaborate on three different means of deploying Koko and discuss the relative merits of each.

1.3 Kokomo Development Methodology

Regardless of Koko’s technical benefits it will not be useful if developers find it too difficult or obtrusive. To this end we have designed a methodology called Kokomo that steps the developer through a process for configuring Koko and incorporating it into an application. The methodology can support both personal (single agent) and social (multiagent) applications. Interestingly, Kokomo is based on the notion of expressives, which as mentioned above are an important but little-known (especially in the agents community) class of communicative acts. Further, Kokomo’s application of expressive communicative acts is the first such methodology to our knowledge.

Using this methodology application developers can construct applications that are both social and affective. As such, the combination of the Koko middleware and this methodology
enable AOSE to expand into the design and creation of affective social applications. The methodology’s utility extends beyond that of social applications as it is also designed to be applied in single agent environments by capturing the expressive interactions between the user and software.

Additionally, in Section 4.4 we detail a developer study, that we conducted to determine the actual and perceived benefits of Koko and Kokomo to application developers. Our hypothesis is simple: developers who use Koko and Kokomo can more easily construct an affective application than those who do not, while at the same time not diminishing the quality of their application. The study consisted of the same application assigned to groups of developers employing Kokomo (with Koko), Koko alone, and neither (just traditional techniques).

Our evaluation measured the ease of constructing an application both subjectively and objectively. We collected subjective developer feedback on their perceived difficulty via surveys, which we collected throughout the duration of the study. The feedback indicates a lower perception of difficulty for the affective portions of the assignment when using Kokomo than without it. An analysis of objective difficulty measures (code metrics and effort analysis) shows that Kokomo yields the best results on nearly every measure of code complexity and effort.

### 1.4 Summary of Contributions

Individually our contributions can be categorized as Koko, Kokomo, and socio-affective applications. However, these items are intertwined together and to understand the full scope of our contribution we must take a holistic view.

The foundation of our contribution is the Koko architecture. When compared to existing approaches for constructing affective applications, the benefits of the Koko architecture are an increase in developer productivity, an enhanced user experience, and the enabling of affective social applications.

Our contribution of the Kokomo methodology depends on the existence of Koko. The methodology guides developers through the process of creating an affective application using
Koko in terms that developers without training in affect modeling can understand. Conversely, Koko also benefits from Kokomo since without the methodology the benefits of the architecture may not be fully realized. To prove the utility of our contribution, we validated the real and perceived benefits of Kokomo (and by extension Koko) to software developers through a developer study.

The combination of Koko and Kokomo produces our final contribution, socio-affective applications. Koko is the first architecture designed to support both inter and intra-application communication of affect. By enabling such interactions we open the door for software developers to create applications that are not only affective but also social. These socio-affective applications can embed users in the real world by providing a user experience that is both personal and emotional.

It is important to note that our work does not introduce any new theories or paradigms for the field of affective computing, but rather seeks to employ existing affective theories in a compelling way. The main contributions of this work are focused on software engineering as we seek to expand the scope of agent-oriented architectures and methodologies into the realms of expressive communication and affective agents. Additionally, unlike many agent-oriented methodologies, Kokomo has been empirically evaluated via a developer study.

1.5 Organization

The remainder of this document is arranged as follows. The background chapter reviews the existing research that our work is built upon, namely appraisal theory models and communicative acts, as well as the progress of modern affective gaming and mobile social applications. The following three chapters each cover a significant portion of our research. The architecture chapter provides detailed description of the components that compose Koko, a formal description of Koko’s interfaces, and some possible deployment strategies. The methodology chapter describes the Kokomo methodology, shows an example of its usage, and empirically evaluates it’s benefit to developers. The applications chapter describes two socio-affective applications
built using both the architecture and methodology. Finally, the discussion chapter addresses some of the implications of Koko and the future work that we plan to pursue.
Chapter 2

Background

This chapter provides the background material needed to understand the technical underpinnings of Koko and the current industry trends that motivated us to develop it. The first two sections of the chapter provide a synopsis of two areas that are fundamental to both Koko and Kokomo: (1) appraisal theory as a foundation for modeling affect and (2) communicative acts and their relevance to AOSE. The final two sections of the chapter describe trends in two different industries (i.e., gaming and mobile) that influenced the design of our systems and also provided motivation for building them.

2.1 Appraisal Theory

Smith and Lazarus’ [46] cognitive-motivational-emotive model, the baseline for current appraisal models (see Figure 2.1), conceptualizes emotion in two stages: appraisal and coping. *Appraisal* refers to how an individual interprets or relates to the surrounding physical and social environment. An appraisal occurs whenever an event changes the environment as interpreted by the individual. The appraisal evaluates the change with respect to the individual’s goals, resulting in changes to the individual’s emotional state as well as physiological responses to the event. *Coping* is the consequent action of the individual to reconcile and maintain the environment based on past tendencies, current emotions, desired emotions, and physiological responses [26].
Koko focuses on a section of the appraisal theory process (denoted by the dashed box), because Koko is intended to model emotions in human subjects. As a result, the other sections of the process are either difficult to model or simply outside of Koko’s control. For instance, the coping section of the process is imperative for modeling emotions for virtual characters since their entire emotional life cycle is controlled by software [52]. However, when the subject is outside the control of software a model of coping serves only as a predictor of how the subject may react since the actual outcome is ultimately determined by the subject. Therefore, we have chosen not to include an explicit model of coping within Koko as its inclusion would greatly increase Koko’s complexity and scope while offering little benefit to the developers using Koko. We however do not discount the usefulness of predicting the user’s future emotional responses and address the matter in the latter part of Section 3.2.1.

A situational construal combines the environment (facts about the world) and the internal state of the user (goals and beliefs) and produces the user’s perception of the world, which then drives the appraisal and provides an appraisal outcome. This appraisal outcome is made up of multiple facets, but the central result is “Affect” or current emotions. For practical purposes, “Affect” can be interpreted as a set of discrete states with an associated intensity. For instance,
the result of an appraisal could be that you are happy with an intensity of $\alpha$ as well as proud with an intensity of $\beta$. Unfortunately, selecting the set of states to use within a model is not an easy task as there is no one agreed upon set of states that covers the entire affective space.

Appraisal theory’s representation of Affect is what we often refer to as a user’s affective state. The user’s affective state is best described as the collection of the emotions that the user is experiencing at a given point in time. Koko represents this psychological concept in a computational unit called an affect vector, which we describe in Section 3.1.1.

Appraisal theory finds broad application because of its computational nature. Its applications include educational games [6], similar to the task in Kokomo’s developer study. We note that affective computing extends beyond our discussion of appraisal theory and Picard [37] provides an excellent overview of the affective computing field which includes a more detailed discussion of appraisal theory.

Next, we look at three existing approaches for modeling emotion based on appraisal theory. The first is a classic approach which provides the foundation for the set of affective states used within Koko. The final two are contemporary approaches to modeling emotion with the primary distinction among them being that EMA focuses on modeling emotions of virtual characters while CARE concentrates on measuring human emotional states.

OCC Ortony, Clore, and Collins [36] introduced the so-called OCC model, which consists of 22 types of emotions that result from a subject’s reaction to an event, action, or object. The OCC’s set of emotions have turned out to cover a broad portion of the affective space. Elliot [13] expanded the set of emotions provided by the OCC to a total of 24 emotions. Indeed, Koko employs this set of 24 emotions as its baseline affective states. Further, the OCC model is effectively realized computationally, thus enabling simulations and real-time computations. Consequently, the OCC model is employed widely as a representation of affect.

EMA The Emotion and Adaptation model leverages SOAR [34] to extend Smith and Lazarus’ model for applications involving nonplaying characters [22]. EMA monitors a virtual character’s
environment and triggers an appraisal when an event occurs. It then draws a set of conclusions or appraisal frames, which reflect the character’s perception of the event. EMA then assigns emotions and emotional intensities to each appraisal frame, and passes on the appraisal frame to the coping mechanism, which decides the best actions to take based on the character’s goals [20]. Further, EMA was one of the first domain independent models of affect [23] and its design helped lay some of the conceptual foundations for Koko. Further, even though EMA differs from Koko in that it is focused on modeling affect in virtual characters, EMA’s insights in the area of domain independence were critical in Koko’s design.

CARE The Companion-Assisted Reactive Empathizer (CARE) supports an on-screen character that expresses empathy with a user based on the affective state of the user [32]. The user’s affective state is retrieved in real-time from a preconfigured static affect model, which maps the application’s current context to one of six affective states. The character’s empathic response is based on these states.

CARE populates its affective model offline. First, a user interacts with the application in a controlled setting where the user’s activity is recorded along with periodic responses from the user—or from a third party—about the user’s current affective state. Second, the recorded data is compiled into a predictive data structure, such as a decision tree, which is then loaded into the application. Third, using this preconfigured model the application can predict the user’s affective state during actual usage.

The probabilistic model used in CARE is well suited for the Koko architecture and is used a reference model throughout the document. Other probabilistic models similar to CARE, such as the one used in the Prime Climb [6] game would also be suitable reference models for Koko. We have chosen CARE as our one of our primary references because it has an easily understandable description and its implementation was readily accessible.
2.1.1 Cognitive-Based User Affect Models

Koko supports all types appraisal theory models, as described earlier in this section. However, Koko’s default implementation provides a particular type of affect models known as cognitive-based user affect models (CB-AUM) [30]. The CARE model is a type of CB-AUM model. A CB-AUM model relies heavily on machine learning techniques. The basic flow of such a model can be conceptualized in the following steps.

1. Seed the model with information about the user’s environment. This seeding is based on the application-specific configuration.

2. Provide training data. Either the user or someone observing the user must record the user’s emotional state in conjunction with data from the application or the sensors. This is simplest if the application itself is appropriately instrumented, e.g., to query the user for their emotions. It can optionally be acquired offline.

3. Learn a model from the training data. Koko does this using the Weka toolkit. In general, user affect modeling [30] makes heavy use of various machine learning techniques to adapt to the behavior patterns of individual users.

4. Provide probabilistic predictions regarding the user’s affective state.

2.2 Communicative Acts

The philosophers Austin [2] and Searle [42] developed speech act theory founded on the principle that language is an action. In other words, when an agent communicates it alters the state of the world. Communicative acts are grouped based on their effects on the agent’s internal state or social relationships. Specifically, assertive acts are intended to inform, directive acts are used to make requests, and expressive acts allow agents to convey emotion.

Existing agent communication languages and methodologies disregard expressives. AOSE methodologies specify messages at a high level and therefore are not granular enough extract
the meaning of the messages [11]. On the other hand, agent communication languages specify messages at the appropriate level of detail, but omit expressives. Instead, they have focused on other communicative acts, such as assertives and directives, which can be readily incorporated into traditional agent BDI frameworks [55].

2.3 Evolution of Affective Gaming

As we look into the future of affective game development, it is beneficial to look where the field has come from. In this section, we walk through a series of illustrations that provide a high-level look at the progression of the programming model from traditional to affect-aware games. These programming models are not representative of all the advances in gaming, but rather are to be understood as simplified schemas that serve as a means by which we can illustrate the benefits of continued improvements.

![Monolithic programming model](image-url)

**Figure 2.2: Monolithic programming model**

**Monolithic.** In a game that subscribes to the monolithic programming model, the developer is responsible for all facets of the game’s development. Figure 2.2 shows an application whose logic includes within it a rich variety of functionalities needed to make the application work. Whereas this level of fine-grained control may be desirable in some situations, applications that
subscribe to this model suffer from numerous shortcomings. Rollings and Morris [40] address a variety of these shortcomings and we outline a few of the more critical ones in the following paragraphs.

Perhaps the most critical shortcoming is that a developer needs to have expertise in all aspects of the game. For example, in games where the player operates in three-dimensional space, the developer needs to have the skills required to support a variety of subtle features such as physics, lighting, and 3D rendering. To build a state-of-the-art game you need expertise in many areas and it often becomes impractical to assemble a development team with the required skill set.

Two other prominent shortcomings arise in connection with reusability and maintainability. A game developed using the monolithic model most likely incorporates components of the game that are applicable in other games. Porting the largely common source code between games results in code duplication and increases the chances of introducing new bugs into the software. This copy-and-paste approach to software development becomes more problematic over time as the developer must then support more and more similar versions of the duplicated code with each new game that is introduced.

![Component-based programming model](image)

**Figure 2.3:** Component-based programming model
Component-based. The component-based programming model addresses many of the issues posed by the monolithic model at the cost of total control [40]. As Figure 2.3 shows, in a component-based model, the developer is responsible only for the application logic while the remaining aspects of the game are abstracted by a third-party toolkit (that is, a game engine). The various components that are abstracted out become reusable and more easily maintainable than in the monolithic model. For example, by using a game engine, the developer does not need to develop a new physics engine for each game, but instead may just configure the engine to enforce the appropriate physics laws for each game.

Abstracting as much of the generic aspects of a game into the game engine enables the developer to focus on the creative aspects of their game. The developer then only needs to configure the game engine to customize game-specific features. This reduces the expertise needed by the developer in a given field. Going back to our physics illustration, the developer does not need to know how to write the software to simulate the physical properties of objects, but instead only needs to know how to configure the physics engine to obey certain physical laws. Moreover, the configuration task is often made easier for the developer by the engine providing defaults for certain game types (for example, physical properties of the Earth versus the Moon) thereby eliminating the need for any complex configuration.

The component-based programming model reflects the state of the art in software development practices. Therefore, it is the model of choice in today’s major game titles. Two of the leading game engines are the Unreal Engine [15] and the Source Engine [54], both of which are designed for use on personal computers and video game consoles.

Affect-Aware. As we discussed in Section 1.2.1, there has been an emergence of a new gaming genre, which we call affect-aware games. Although the introduction of affect into the programming model does not dictate a specific programming model, it has been common practice to use a modified version of the component-based programming model. The main modification is the introduction of the player’s affect model into the application logic. Examples of this affect-aware programming model in action are the games Treasure Hunt [32], Crystal
Island [27], and FearNot [12].

Employing a model similar to the component-based model enables the developers of these affect-aware games to build compelling games. Further, by abstracting out irrelevant details, it enables the developer to focus on the “primary” (from the agents standpoint) component of their game, namely, the affect model. Current affect-aware games must be built by developers who are experts in the field of affect modeling. The affect model in these games is tightly coupled to the application logic making it necessary for the developer to understand the intricacies of the model in order to construct the game.

Further, the player’s environment extends beyond the virtual world and in order to more accurately predict the user’s state many affective games incorporate physical sensors. These sensors may either be used solely by the affect model or by a combination of the affect model and the application logic. For example, in many cases, a heart rate sensor may be only relevant to the affect model, but in a physical fitness game it would be logical to also display the heart rate information directly to the user.

### 2.4 Brief History of Mobile Social Applications

Table 2.1 provides a high-level overview of the progression of mobile, social applications. It breaks the progression into four distinct categories that are representative of different families
of social applications. The *basis* column denotes what the social relationship is founded on and the *user model* column denotes how the application customizes itself to a user’s needs.

Table 2.1: Types of mobile social applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>World</th>
<th>Basis</th>
<th>User Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>Email &amp; IM</td>
<td>Virtual</td>
<td>Buddy List</td>
<td>Static</td>
</tr>
<tr>
<td>Networking</td>
<td>Facebook</td>
<td>Virtual</td>
<td>Social Graph</td>
<td>Static</td>
</tr>
<tr>
<td>Geo-Social</td>
<td>Loopt</td>
<td>Virtual + Real</td>
<td>Spatial</td>
<td>Predictive</td>
</tr>
<tr>
<td>Socio-Affective</td>
<td>booST</td>
<td>Virtual + Real</td>
<td>Emotional</td>
<td>Predictive</td>
</tr>
</tbody>
</table>

**Traditional** social applications are those like email and instant messaging that were among the first to be ported to mobile devices. These applications operate completely within a virtual environment and typically offer users ways to exchange information, including audio and video.

**Networking** was a logical next step in order to make virtual interactions more interesting through an explicit, sharable representation of a user’s social circle. Networking appli-
cations offer ways for users to discover new connections, and can support a variety of interactions (e.g., picture sharing, status updates, messaging, and games) in a cohesive manner.

**Geo-Social** applications combine social networks with a user’s physical environment, typically using sensors on a mobile device to gather information about the user’s surroundings, which they then relate to relevant content found within the user’s social network. Additionally, by incorporating real-world data, they can create more complex and dynamic models of user preferences and behaviors. Services such as Loopt exemplify this model because they recommend nearby restaurants and social events based on the user’s preferences, whether explicitly acquired or learned from previous interactions.

**Socio-Affective** applications are what we introduce in Chapter 5. Like geo-social applications, they rely upon a user’s social network and exploit knowledge of the user’s surroundings. In addition, they incorporate emotional awareness and support personalizing the user experience based on a participant’s emotional state.
Chapter 3

Architecture

Existing affective applications tend to be monolithic where the affective model, external sensors, and application logic are all tightly coupled. As in any other software engineering endeavor, monolithic designs yield poor reusability and maintainability. Similar observations have led to other advances in software architecture [43]. Most pertinently, the idea of a knowledge base being separate from a solution strategy is motivated on such grounds.

Koko promotes a loosely coupled, service oriented style of architecture where the affect model, application logic, and external sensors are individual units. The primary role of Koko in this architecture is to act as both an affect model container and as an agent communication middleware. Koko’s affect models focus on a section of the appraisal theory process (denoted by the dashed box in Fig. 2.1) in which the models absorb information about the agent’s environment and produce an approximation of the agent’s affective state. Koko then enables that affective information to be shared with the corresponding application and among other agents. It is important to note that since Koko is designed to model human users, the environment of the agent extends into the physical world. To better report on that environment, Koko supports input from a variety of physical sensors (e.g. GPS devices).

As an agent communication middleware Koko promotes the sharing of affective information at two levels: cross-user or inter-agent communication and cross-application or intra-agent
communication. For a social (multi-agent) application, Koko enables agents to communicate via expressives. Expressives enable agents to share their current affective state among other agents within Koko (the format of an expressive message is outlined in Section 4). Koko also provides a basis for applications—even those authored by different developers—to share information about a common user. This is simply not possible with current techniques because each application is independent and thereby unaware of other applications being employed by a user.

3.1 Main Components and their Usage

Figure 3.1 shows Koko’s general architecture using arrows to represent data flow. The following subsections provide details on each of Koko’s major components. Then after we explain the groundwork of the architecture we elaborate on the formal interface definitions in the following section.

![Figure 3.1: Koko architectural overview](image)

Koko hosts an active computational entity or agent for each user. In particular, Koko requires and ensures that there is exactly one agent per human, who may use multiple Koko-based applications. (In principle, a user may open multiple Koko accounts and thus decouple
his or her applications from each other, but doing so would reduce the benefit of Koko to the user.) Each user agent in Koko may access global resources such as sensors and messaging but in all other respects operates autonomously with respect to the other agents.

Prior to describing each component individually it is beneficial to understand how the components interact as a whole. Using Figure 3.1 as our reference, we demonstrate how information flows from the application through Koko. The application first interacts with Koko’s user manager to obtain a reference to the agent that represents the application’s current user. The application then updates the agent with events occurring in the application, which are stored in the event repository. The agent combines the information from the application with additional sensory information and performs an appraisal of the user’s current emotional state using the affect model instance stored in the affect model container. The appraisal generated by the model produces a vector of emotions that is made available to the application as well as a cross-application mood model. The mood model combines the vectors produced by all the agent’s applications and produces a mood vector. The application then retrieves the user’s current emotional state or mood from the affect repository and uses the information to alter gameplay. Finally, as a convenience the application can request direct access to a sensor’s output via the sensor manager.

3.1.1 The Affect Model Container

This container manages one or more affect models for each agent. Each application specifies exactly one affect model, whose instance is then managed by the container. As Figure 3.2 shows, an application’s affect model is specified in terms of the affective states as well as application and sensor events, which are defined in the application’s configuration (described below).

Configuring the application’s affect model at runtime enables us to maintain a domain-independent architecture while supporting domain-dependent affect model instances. Further, in cases such as CARE, we can construct domain-independent models (generic data structures) provide the domain context at runtime (object instances). This is the approach we take in
Koko by constructing a set of generic affect models that it provides to applications. These generic affect models follow CARE’s supervised machine learning approach of developing an affect model by populating predictive data structures with affective knowledge. These affect models are built, executed, and maintained by the container using the Weka toolkit [57]. The two standard affect models that Koko provides use Naive Bayes and decision trees as their underlying data structures.

To accommodate models with drastically different representations and mechanisms, Koko encapsulates them via a simple interface. The interface takes input from the user’s physical and application environment and produces an affect vector. The resulting affect vector contains a list of elements, each of which corresponds to a emotion that the user is currently feeling. Thus the affect vector can be described as Koko’s probabilistic representation of the user’s affective state at a given moment. The emotions contained within an affect vector are selected from an ontology that is defined and maintained via the developer interface vocabulary. Using this ontology, each application developer selects the emotions to be modeled for their particular application. For each selected emotion, the vector includes a quantitative measurement of the emotion’s intensity. The intensity is a real number ranging from 0 (no emotion) to 10 (extreme emotion).
3.1.2 Mood Model

Following EMA, we take an emotion of a specific user as the outcome of one or more specific events and a mood for a specific user as a longer lasting aggregation of the emotions of that user. An agent’s mood model maintains the user’s mood across all applications registered to that user.

Koko’s model for mood is simplistic as it takes as input one or more affect vectors and produces a mood vector, which includes an entry for each emotion that Koko is modeling for that user. Each entry represents the aggregate intensity of the emotion from all affect model instances associated with that user. Consequently, if Koko is modeling more than one application for a given user, then the user’s mood is a cross-application measurement of the user’s emotional state. For example, suppose a user has two Koko-based applications, which model the emotions of pride and shame and of pride and fear, respectively. The resulting mood vector for that user would contain three entries namely pride, shame, and fear.

Koko provides a default mood model implementation in which the value of each mood vector entry is determined using the weighted average for all affect vectors associated with the user. To ensure that a user’s mood is up to date with respect to recent events, we follow Picard [37] and introduce a mood decay formula as a way to reduce the contribution of past events. The decay formula reduces the effect that a given emotion has on the user’s mood over time on a logarithmic scale that gives preference to the most recent emotions. We further augment our mood model with the concept of mood intensity from Dias and Paiva’s work on the FearNot! model of emotions [12]. The mood intensity sums all positive and negative emotions that make up the user’s current mood. The mood intensity helps us determine the degree to which a positive or negative emotion impacts a specific user. For example, if a user has a positive mood intensity then a slightly negative event may be perceived as neutral, but if the event were to recur, the user’s mood intensity would continue to degrade, thereby amplifying the effect of the negative event on the user’s mood over time.
3.1.3 Affect Repository

The affect repository is the gateway through which all affective data flows in Koko. The repository stores both affect vectors (application specific) and mood vectors (user specific). These vectors are made available both to external applications and to the affect and mood models of the agents. This does not mean all information is available to a requester, as Koko implements security policies for each user (as discussed in connection with the user manager below). Any entity can request information from the repository but the only vectors returned (if any) are those the requester has the permission to access.

The vectors within the repository can be retrieved in one of two ways. The first retrieval method is through a direct synchronous query that is similar to a SQL select statement. The second method enables agents within Koko to register a listener, which is notified when data is inserted into the repository that matches the restrictions provided by the listener. The second method allows for agents to have an efficient means of receiving updates without proactively querying and placing an unnecessary burden on the repository.

3.1.4 Event Repository

With respect to storage, retrieval, and security, the event repository is nearly identical to the affect repository. Instead of storing vectors of emotion, the event repository stores information about the user’s environment. This environmental information is comprised of two parts: information supplied by the application and information supplied by sensors. In either case, the format of the data varies across applications and sensors as no two applications or sensors can be expected to have the same environment. To support such flexibility we characterize the data in discrete units called events, which are defined on a per application or sensor basis. Every event belongs to an ontology whose structure is defined by the developer using the Koko developer interface.


3.1.5 Sensor Manager

Information about a user’s physical state (for example, heart rate and perspiration) as well as information about the environment (for example, ambient light, temperature, and noise) can be valuable in estimating the user’s emotional state. Obtaining such information is a programming challenge because it involves dealing with a variety of potentially arcane sensors. Accordingly, Koko provides a unified interface for such sensors in the form of the sensor manager. This approach yields key advantages. First, a single sensor can be made available to more than one application. Second, sensors can be upgraded transparently to the application and affect model. Third, the overhead of adding sensors is amortized over multiple applications.

The sensor manager requires a plugin for each type of sensor. The plugin is responsible for gathering the sensor’s output and processing it. During the processing stage, the raw output of the sensor is translated into a sensor event whose elements belong to the event ontology in the developer interface vocabulary. This standardized sensor event is then provided as input to the appropriate affect models.

The sensor plugin is responsible for processing data, but does not make assumptions about the affective state of the user. For example a facial recognition sensor could process an image of the user’s face. The sensor could then output data about the user’s facial features such as the position of the cheekbones, dilation of the eyes, and it could even go as far as interpreting the expression as a smile, but it will not make an assumption on what that data means. The data is simply made available to the affect model, which is ultimately responsible for generating the approximation of the user’s affective state. In practice, it is expected that the affect model will either have a set of predefined rules for how to interpret the sensory data or will employ some form of machine learning to understand the data.

3.1.6 User Manager

The user manager keeps track of the agents within Koko and maintains the system’s security policies. The manager also stores any information provided by the user on behalf of the agent.
This includes information such as which other agents have access to the affective data stored by the given agent and which aspects of that data they are eligible to see. It is the user manager’s responsibility to aggregate that information with the system’s security policies. Further, the user manager provides the resulting security restrictions to the sensor manager and the affect repository. Privacy policies for storing and sharing such personalized information are crucial for such applications [39], but the details of such policies lie outside the scope of this paper.

3.1.7 Developer Interface Vocabulary

Koko provides a vocabulary through which the application developer interacts with Koko. The vocabulary consists of two ontologies, one for describing affective states and another for describing the environment. The ontologies are encoded in OWL (Web Ontology Language). The ontologies are designed to be expandable to ensure that they can meet the needs of new models and applications.

The ontologies ensure that data produced by one architectural component can be understood by any other architectural component or subscribing application. The ontologies are not universal since each implementation of the architecture may extend the default ontologies. There are efforts towards standardization, such as the W3C Emotion Incubator Group (http://www.w3.org/2005/Incubator/emotion/), and if such an standard were created it could be merged into the existing ontology.

The emotion ontology describes the structure of an affective state and provides a set of affective states that adhere to that structure. Koko’s emotion ontology by default provides the 24 emotional states proposed by Elliot [13]. Those emotions include states such as joy, hope, fear, and disappointment.

The event ontology can be conceptualized in two parts: event definitions and events. An event definition is used by applications and sensors to inform Koko of the type of data that they will be sending. The event definition is constructed by selecting terms from the ontology that apply to the application, resulting in a potentially unique subset of the original ontology. Using
the event definition as a template, an application or sensor generates an event that conforms to the definition. This event then represents a view of the state of the application or sensor at a given moment. When the event arrives at the affect model it can then be decomposed using the agreed upon event definition.

![Event ontology example](image)

Koko comes preloaded with an event ontology (partially shown in Figure 3.3) that supports common contextual elements such as time, location, and interaction with application objects.

Consider an example of a user seeing a snake, which may have interesting consequences of a user’s emotions—an ordinary person may be fearful whereas a herpetologist may be joyful. To describe this event for Koko you would create an event seeing, which involves an object snake. The context of such an event is often extremely important. For example, the user’s emotional response could be quite different depending on whether the location was in a zoo or the user’s home. Therefore, the application developer should identify and describe the appropriate events (including objects) and context (here, the user’s location).

### 3.2 Formal Description of Koko

Now that we have laid the groundwork, we describe the Koko architecture in more formal terms. The description is divided into two segments, with the first describing the interfaces used by a game developer and the second describing the interfaces used by an affect model.
designer. Our motivation in presenting these interfaces conceptually and formally is to make the Koko architecture *open* in the sense of specifying the interfaces declaratively and leaving the components to be implemented to satisfy those interfaces.

Before we begin to formally define the interfaces we must first introduce the notation that will be used. The interfaces are specified using the mathematical notation language *Z* [51] with the additional convention of schema encapsulation [1] for clarity. We initially define a set of primitives and objects that we use in the interface description. The objects defined in the schema (for example, KokoAgent) are formal representations of the entities described in Section ??_. These objects are not the complete representation of those entities but rather only the necessary components required by the interface. Further, for clarity we briefly describe the primitives, followed by the *Z* notation for both the primitives and the objects.

- **AGENT_ID**: unique identifier for the agent and user represented by that agent
- **APPLICATION_ID**: unique identifier for a given application
- **AFFECT_MODEL**: model used to compute the user’s affective state
- **AFFECT_INTENSITY**: a quantitative measurement of an emotion’s intensity
- **AFFECTIVE_STATE**: a single state selected from the emotion ontology
- **EVENT**: an event instance as defined by the event ontology
- **VECTOR_TYPE**: distinguishes an affectVector from a moodVector

Here is a brief introduction to some of the identifiers and operators defined in *Z*. A value in all capital letters is a primitive type. An identifier with a trailing “?” indicates an input, where “!” indicates an output. The symbol \( \mathbb{P} \) is used to represent the power set. In interface definitions, “\( \Delta \)” is used to denote an operation that will update the schema, while “\( \Xi \)” is used to indicate the operation will not update the schema’s state. Further, an identifier with a trailing
' (prime) indicates the value of the identifier after the schema’s state has been updated.

\[ \text{AGENT\_ID, APPLICATION\_ID, AFFECT\_MODEL, AFFECT\_INTENSITY, AFFECTIVE\_STATE, EVENT} \]

\( \text{VECTOR\_TYPE ::= application | mood} \)

\begin{verbatim}
KokoAgent
  agentID : AGENT\_ID
  moodVector : AffectVector
  applications : \( \mathbb{P} \) KokoApplication
  getApplication : APPLICATION\_ID \( \rightarrow \) KokoApplication
  applications \( \neq \) \( \emptyset \)
  moodVector.type = mood

KokoApplication
  applicationID : APPLICATION\_ID
  affectModel : AFFECT\_MODEL
  affectVectors : \( \mathbb{P} \) AffectVector
  events : \( \mathbb{P} \) EVENT
  predictOutcome : EVENT \( \rightarrow \) AffectVector
  affectModel \( \neq \) \( \emptyset \)
  \( \forall v : \) affectVectors \( \bullet v\text{-type} = \) application
\end{verbatim}
3.2.1 Application Developer Interfaces

The interfaces that enable a developer to couple Koko with other modules are naturally broken into two distinct categories: configuration and runtime. The configuration interfaces enable developers to notify Koko of their intent to use the middleware and also agree to the format of the runtime communication. The runtime interfaces are then used to send information needed by the affect model and to retrieve the user’s affective state or an approximation thereof.

Application Runtime Interface

The application runtime interface is broken into two discrete units, namely, event processing and querying. Before we look at each unit individually, it is important to note that the contents of the described events and affect vectors are dependent on the application’s initial configuration, which is discussed in Section 3.2.1.

Application Event Processing. The express purpose of the application event interface is to provide Koko with information regarding the application’s environment. During configuration, a developer defines the application’s environment via the event ontology in the developer interface. Using the event ontology, the application encodes snapshots of its environment as events and passes them to Koko for processing. The formal description of this interaction is as follows.
Upon receipt, Koko stores the event in the agent’s event repository, indexed by time of occurrence, where it is available for retrieval by the appropriate affect model. This data combined with the additional data provided by external sensors provides the affect model with a complete picture of the user’s environment.

**Application Queries.** Applications are able to query for and retrieve two types of vectors from Koko. The first is an application-specific affect vector and the second is a user-specific mood vector, both of which are modeled using the developer interface’s emotion ontology. The difference between the two vectors is that the entries in the affect vector are dependent upon the set of emotions chosen by the application when it is configured, whereas the mood vector’s entries are an aggregation of all emotions modeled for a particular user.

When the environment changes, via application or sensor events, the principles of appraisal theory dictate that an appraisal be performed and a new affect vector computed. The resulting vector is then stored in the affect repository. The affect repository exposes the stored vector to an application via two interfaces, each of which we describe formally next.
\[\text{GetCurrentAffectVector}\]

\(\Xi \text{KokoAgent}\)

\(\text{userID? : AGENT\_ID}\)

\(\text{applicationID? : APPLICATION\_ID}\)

\(\text{affectResult! : AffectVector}\)

\(\text{userID?} = \text{agentID}\)

\((\text{let application} \equiv \text{getApplication(} \text{applicationID?) \cdot \text{getApplication(} \text{applicationID?)} \neq \emptyset\})\)

\((\text{let maxTime} \equiv \text{max}\{\forall v.\text{time : application.affectVectors}\})\)

\(\text{affectResult!} = \{v : \text{application.affectVectors} | v.\text{time} = \text{maxTime}\}\)

\[\text{GetAffectVector}\]

\(\Xi \text{KokoAgent}\)

\(\text{userID? : AGENT\_ID}\)

\(\text{applicationID? : APPLICATION\_ID}\)

\(\text{startTime? : Z}\)

\(\text{endTime? : Z}\)

\(\text{affectResults! : } \mathbb{P} \text{AffectVector}\)

\(\text{userID?} = \text{agentID}\)

\((\text{let application} \equiv \text{getApplication(} \text{applicationID?) \cdot \text{getApplication(} \text{applicationID?)} \neq \emptyset\})\)

\(\text{affectResults!} = \{v : \text{application.affectVectors} | v.\text{time} \geq \text{startTime?} \land v.\text{time} \leq \text{endTime?}\}\)
Additionally, an application can pass in a contemplated event and retrieve an affect vector based on the current state of the model. The provided event is not stored in the event repository because it is treated as not having occurred and thus does not update the state of the model. This enables the application to compare potential events and select the event that it can safely expect to elicit the best emotional response from the user. The interface is formally defined as follows.

\[
\text{GetPredictedAffectVector} \quad \Xi \text{KokoAgent} \\
\text{userID}? : \text{AGENT\_ID} \\
\text{applicationID}? : \text{APPLICATION\_ID} \\
\text{predictedEvent}? : \text{EVENT} \\
\text{affectResult}! : \text{AffectVector}
\]

\[
\text{userID}? = \text{agentID} \\
(\text{let application} == \text{getApplication}(\text{applicationID}?) \bullet \\
\text{getApplication}(\text{applicationID}?) \neq \emptyset) \\
\text{affectResult}! = \text{application.predictOutcome}(\text{predictedEvent})
\]

Mood vectors, unlike affect vectors, aggregate emotions across applications. As such, a user’s mood is relevant across all applications. Suppose a user is playing a game and becoming frustrated playing it. Suppose the game’s affect model recognizes this. The user’s other affect-enabled applications can benefit from the knowledge that the user is frustrated even if they cannot infer that it is from a particular game. Such mood sharing is natural in Koko because it maintains the user’s mood and can potentially supply the mood to any application that may request it. The following formalizes the above mechanism for retrieving the mood vector.
GetMood
ΞKokoAgent

userID? : AGENT_ID
moodResult! : AffectVector

userID? = agentID
moodResult! = moodVector

Application Configuration Interface

Properly configuring an application is key because its inputs and outputs are vital to all of
the application interfaces within Koko. In order to perform the configuration, the developer
must first gather key pieces of information and next supply that information to Koko using the
following interface:

[SENSOR_ID, MODEL_ID]
Additional Z primitives are needed to describe the configuration, there descriptions are as follows. The affectiveStates are the states (drawn from the emotion ontology) that the application wishes to model. The eventDefinitions describe the structure (created using the event ontology) of all necessary application events. The game developer can encode the entire application state using the ontology, but this is often not practical for large games. Therefore, the developer must select the details about the application’s environment that are relevant to the emotions they are attempting to model. For example, the time the user has spent on a current task will most likely affect the user’s emotional status, whereas the time until the application needs to garbage collect its data structures is most likely irrelevant to the user. The sensorIDs
and modelID both have trivial explanations. Koko maintains a listing of both the available sensors and affect models, which are accessible by their unique identifiers. The developer must simply select the appropriate sensors and affect model and record their identifiers.

Further, Koko enables affect models to perform online, supervised learning by classifying events via a set of emotions. Applications can query the user directly for the user’s emotional state and subsequently pass that information to Koko. In general, many applications include well-defined places where they can measure their user’s responses in a natural manner, thereby enabling the online learning of affective state. Applications that do exercise Koko’s learning interface can benefit from improved accuracy in their affect model as they learn more about the user. The formal definition of this interface is as follows. Notice there is no output because this interface is used only to update Koko’s data structures, not to retrieve information.

```
ModelTraining

ΔKokoAgent

userID? : AGENT_ID

applicationID? : APPLICATION_ID

applicationEvent? : EVENT

classifier? : AffectVector

userID? == agentID

(let application == getApplication(applicationID?) •
  getApplication(applicationID?) ≠ ∅)

application.affectModel' = application.affectModel ∪ \{(applicationEvent?, classifier?)}
```

### 3.2.2 Affect Model Interface

Koko supports the addition of new affect models via plugins. An affect model plugin consists of a structured description of the plugin’s contents and a executable unit containing the unique
logic specific to that plugin. The lifecycle of a plugin is straightforward. To start, the plugin is
generated by a third-party provider. The provider then registers the plugin with Koko, which
adds it to the appropriate resource pool. The agents operating within Koko are then able to
select the appropriate plugin from the resource pool. The plugin remains in the resource pool
until explicitly removed by the developer or administrator.

The affect models are the primary reasoning component of a Koko agent, in essence they
form the heart of each agent. Given the importance of the affect model, Koko offers a great deal
of freedom in how an affect model designer implements their model. To enable a wide variety
of models Koko places only one precondition on their construction. The condition is that the
affect models must follow the principles of appraisal theory in so much that they compute the
user’s affective state based on inputs from the user’s environment. To enable this behavior, the
model is provided two standardized means by which to obtain input, events from both sensors
and the application.

We describe the composition and interfaces of an affect model plugin using the same Z
notation that was used to describe the application interfaces. We do not attempt to describe
all the interfaces in this section, but only those that are necessary to construct a basic affect
model. The AffectModelDescription schema contains all the information needed by Koko to
register the plugin. The schema introduces one new primitive, called MODEL_LOGIC. The
MODEL_LOGIC primitive represents the logic responsible for reasoning about the environment
and producing an affect vector. We do not attempt to define this primitive in order to enable
the model designers to build reasoners based on various psychological theories and using their
own custom data structures. Further, we do not presuppose that a given model can support
recognizing and reasoning about all types of emotions and environments. To enable an affect
model designer to inform Koko of it’s limitations they can describe the inputs that they can
accept using the developer interface vocabulary. Specifically the emotion ontology is used for
describing the supportedAffectiveStates and the event ontology for supportedEventDefinitions.
The modelID is used to encode the developers not only the name of the model but also a
description of the model, version number, and other auxiliary information.

\[\text{[MODEL\_LOGIC]}\]

\[
\begin{align*}
\text{AffectModelDescription} & \\
\quad \text{modelID} & : \text{MODEL\_ID} \\
\quad \text{supportedAffectiveStates} & : \mathbb{P} \text{AFFECTIVE\_STATE} \\
\quad \text{supportedEventDefinitions} & : \mathbb{P} \text{EVENT} \\
\quad \text{logic} & : \text{MODEL\_LOGIC} \\
\end{align*}
\]

\[
\begin{align*}
\text{modelID} & \neq \emptyset \\
\text{logic} & \neq \emptyset \\
\end{align*}
\]

Applications choose which affect model to use based on the available \textit{AffectModelDescriptions}. Then agents whose users employ that application use the description to generate an \textit{AffectModelInstance}. The resulting \textit{AffectModelInstance} replaces the \texttt{AFFECT\_MODEL} primitive defined earlier in the section. The difference between the \textit{AffectModelDescription} and \textit{AffectModelInstance} is the same as the difference in programming between an object class and an instance of that class. While that difference is easy to describe verbally, the notation in \texttt{Z} is quite verbose and convoluted. Therefore, for the sake of clarity we state the conversion as follows. The conversion from the affect model description to instance occurs by instantiating the \texttt{MODEL\_LOGIC} using the application’s \textit{affectiveStates} and \textit{eventDefinitions} (like parameters of an object’s constructor), which then produces the \texttt{performAppraisal} function.
Given our precondition that the model operates according to appraisal theory, the model is required to support the Appraisal interface. When the agent’s environment changes an event is generated which compels the agent to conduct an appraisal and produce an affective response. The formal notation for such an interaction is as follows.
The affect model can also support additional non-mandatory interfaces. For instance, it can choose to support affect prediction in order to enable the \texttt{GetPredictedAffectVector} interface or if the model supports learning it can implement the \texttt{ModelTraining} interface. It should also be noted that affect models have access to all its parent agent’s resources and inter-agent messaging capabilities as described in the beginning of this chapter.

### 3.3 Architecture Evaluation

A software architecture is motivated not by functionality but by so-called “ilities” or nonfunctional properties [17]. The properties of interest here—reusability, extensibility, and maintainability—pertain to gains in developer productivity over the existing monolithic approach. In addition, by separating and encapsulating affect models, Koko enables sharing affective data among applications, thereby enhancing user experience. Thus we consider the following criteria.

**Reusability.** Koko promotes the reuse of affect models and sensors. By abstracting sensors via a standard interface, Koko shields model designers from the details of how to access various sensors and concentrate instead on the output they produce. Likewise, application developers can use any installed affect model.

**Maintainability.** Koko facilitates maintenance by separating the application from the affect model and sensors. Koko supports upgrading the models and sensors without changes to the components that use them. For example, if a more accurate implementation of a heart rate sensor becomes available, you could simply unregister the corresponding old sensor and register the new sensor using the old sensor’s identifier. Any model using that sensor would begin to receive more accurate data without any additional programming effort on the part of the application developer. Likewise, a new affect model may replace an older model in a manner that is transparent to all applications using the original model.
**Extensibility.** Koko specifies generic interfaces for applications to interact with affect models and sensors as well as providing a set of implemented sensors and models. New sensors and models can be readily installed as long as they respect the specified interfaces.

**User Experience.** Koko promotes sharing at two levels: cross-application or intra-agent and cross-user or inter-agent communication. For a social or multi-player application Koko enables users or rather their agents to exchange affective states. Koko also provides a basis for applications—even those authored by different developers—to share information about a common user. An application may query for the mood of its user. Thus, when the mood of a user changes due to an application or otherwise, this change becomes accessible to all applications.

**Koko Implementation**

All Koko-based applications described in this document use the same instance of the Koko middleware. Therefore, Koko maintains only one agent for per user even if the user participates in multiple games. This enables the games to offer a better user experience based on the cross-application benefits of Koko. In particular, each game can use the agent’s cross-application mood vectors to gain a better understanding of the user’s affective state.

Further, all applications use the same type of affect model, namely a CARE style model which is represented as a decision tree. The affect model is loaded into Koko as a plugin which adheres to the *AffectModelDescription* that was outlined in Section 3.2.2. Like CARE our affect model is domain independent and must be configured to handle the domain-specific input from each application. To achieve this flexibility we specify which elements of Koko’s event ontology our model supports and describe that requirement in the *AffectModelDescription*. Therefore, each application that employs this particular affect model must adhere to those requirements or Koko will mark them as incompatible. Practically, the compatibility check occurs during the application’s initial startup when the application registers itself with Koko, selects an affect model, and provides its configuration specification. The configuration specification is composed
of the emotions that the application wishes to monitor as well as its set of event definitions that describe the type of data (constructed using the event ontology) it will pass to Koko. Using that specification Koko checks that the application and affect model are compatible and then creates and configures an instance of the affect model specific to that application and the original user. Each subsequent user of the application is given their own unique affect model instance using the same application specification.

The same style of registration and configuration used by affect models is also used for sensors. The major difference is that sensors can be configured for access by both an application and affect model. There are two sensor plugins installed in our implementation, namely a GPS plugin and a skin connectivity plugin. The GPS plugin receives data from a sensor installed on a user’s mobile device and outputs latitude and longitude coordinates for the user. The skin connectivity plugin receives data from an electrical sensor that can be attached to the user while playing a desktop game and it outputs both the user’s galvanic skin response and heart rate. Finally, it is worthwhile noting that sensor data may not always be current or available. To address the potential lack of data our affect model implementation does include any sensor reading older than one minute when it is performing an appraisal.

**Reconstructing Existing Applications**

In Chapter 5 we will show how new applications, such as booST, can be built using Koko. Now we show how Koko can be used to retrofit an existing affective application, resulting in a more efficient and flexible application.

Treasure Hunt (TH) is a educational game which demonstrates the CARE model [32] wherein the user controls a character to carry out some pedagogical tasks in a virtual world. TH appraises the emotional state of the user based on a combination of (1) application-specific information such as location in the virtual world, user’s objective, and time spent on the task and (2) data from physiological sensors providing information on the user’s heart rate and skin conductivity. TH conducts an appraisal every time the state changes and produces one of six
perceived emotional states is output.

To reconstruct TH using Koko, we begin by abstracting out the sensors. The corresponding sensor code is eliminated from the TH code-base because the sensors are not needed by the application logic. After the sensors have been abstracted out, we select the six emotional states from the emotion ontology that match those already used in TH. The final step is to encode the structure of the application’s state information into application events. This basic format of the state information is already defined, as the original TH logs the information for its internal affect model.

Both the original CARE and enhanced Koko models can be thought of as supporting three phases, as outlined in Table 3.1. The difference between the two architectures is how they choose to realize those phases. For example, in CARE’s version of TH the environmental data and emotional classifiers are written to files. The data is then processed offline and the resulting affect model is injected into the application enabling TH to produce the emotional probabilities.

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gather environmental data and emotional classifiers</td>
</tr>
<tr>
<td>2</td>
<td>Perform supervised ML techniques on the data and classifiers</td>
</tr>
<tr>
<td>3</td>
<td>Given environmental data produce emotional probabilities</td>
</tr>
</tbody>
</table>

Koko improves on the CARE architecture by eliminating the need for the application to keep record of its environmental data and also by performing the learning online. As a result, the Koko-based TH can move fluidly from State 1 to State 3 and back again. For example, if TH is using Koko then it can smoothly transition from providing Koko with learning data, to querying for emotional probabilities, and return to providing learning data. In CARE this sequence of transitions while possible, results in an application restart to inject the new affect model.

This improvement can be compared to an iterative software engineering approach versus
the more rigid waterfall approach. CARE corresponds to the waterfall approach, in that it makes the assumption that you have all the information required to complete a stage of the process at the time you enter that stage. If the data in a previous stage changes you can return to a previous state but at a high cost. Koko follows a more iterative approach by removing the assumption that all the information will be available and ensuring that transitions to a previous state incur no additional cost. As a result, Koko yields a more efficient architecture than CARE.

3.4 Deployment Approaches

Koko is an architecture with open interfaces enabling it to be deployed in a variety of configurations. As we proceeded to deploy Koko for booST and hooPS, we considered multiple deployment strategies. In this section we elaborate on three of these deployment options. We avoid claiming that one deployment strategy is superior to the others, because the optimal solution changes based on the application in question and its target environment. Instead, we discuss the pros and cons of each deployment strategy and suggest certain strategies based on characteristics of the application.

The closed strategy is one in which the applications and Koko were operating on the same closed system. The benefits of this style of deployment is that the owner of the closed system has complete control over Koko and its resources. This level of control enables the owner to guarantee certain quality-of-service metrics such as a maximal latency time between (all or selected) applications and Koko. Further, it allows for privacy guarantees to the end user (that is, Alice and Bob) since the system owns the data that exists either in the application or in the executing Koko instance.

Deployment within a closed system does yield significant disadvantages, however. The primary disadvantage of this approach is the scope of the agent’s within Koko. For example, Alice and Bob operate outside of the system and are free to interact with other software. If Alice were to interact with another system that employs Koko, then she would have two distinct
agents modeling her affective state (one per system). Koko requires that there is only one agent per human. Though this is not a direct violation of our correctness assumptions, two Koko instances both having an agent for Alice can cause potential inconsistencies among Koko’s models for Alice. It does limit the effectiveness of Koko in its social, multiagent functions as well as accurately modeling the user’s cross-application affective state. This strategy enables the developer to control not only the application and Koko, but also the communications pipeline between them. The closed system strategy is recommended for applications that must ensure the privacy of a user’s data.

The *global* strategy is one in which the application and Koko operate on separate, open systems. Further, in this case, Koko can operate as a singleton service and as such, Koko’s agents would be aware of all Koko-based applications for a particular user. The resulting global awareness makes this strategy ideal for social, multiagent applications. Another, more subtle benefit is that the system in which the application is running is not responsible for the processing requirements of Koko. This can have significant impact if the platform (for example, a mobile device) operating the application has limited resources.

Unlike the closed system developers, those who employ the global strategy face the dis-
advantage that they cannot make guarantees about latency. Latency is uncertain because communication between the application and Koko may occur over a network that they do not control. Another disadvantage is that the game developer is most likely not the provider of the Koko service. This requires applications to form a service level agreement with the provider of the Koko service and also that the data stored in Koko on behalf of the application is maintained by a third party.

The open, singleton service strategy is recommended for applications that want to take full advantage of the social and multiagent aspects of Koko and are not restricted by the placement of data on third-party systems. This approach is also recommended for applications operating on systems where the processing and energy requirements posed by Koko might be too demanding.

The hybrid strategy combines the above two. It provides all of the features of the global approach, but also enables applications to operate in a manner similar to that of the closed system. In Figure 3.6 we see Alice and Bob utilizing Koko in the same manner as the global approach. In addition, we introduce Charlie who is operating in a partially closed environment. We define this partially closed environment to contain both the application and a Koko client.
The interactions between the application and the Koko client are identical to their interactions in the closed deployment approach, including the ability to enforce various quality of service metrics. The difference between this approach and the closed system approach is in data storage and agent locality.

When Charlie interacts with the application, his Koko client retrieves his agent from global Koko instance. The agent stores all of the events provided by application and the affect model in local storage and never shares that data with the global Koko instance. The agent does however asynchronously transmit all other information back to the global Koko instance. Further, to maintain a consistent state in the global Koko instance, when an agent is proxied by a Koko client, it cannot be used by another application until it is released by the Koko client.

Figure 3.6: Hybrid system deployment with performance proxy

The benefits of using the Koko Client in a partially closed system is that the developer
is able to enforce quality of service metrics as well as maintain control over all details of the application’s environment. Further, since Charlie exists outside of the partially closed system he is able to user other applications that follow the global approach while still maintaining only one agent within Koko. Using a partially closed system does, however, come with the penalty of having to load the agent from the global Koko instance at startup. It also restricts the end user to only be able to use one application at a time when that application exists in a partially closed system.

The hybrid system approach is recommend for developers who want to leverage the full social and multiagent aspects of Koko, but are concerned about data privacy and quality of service in certain applications. The hybrid approach’s feature set is a superset of that offered by the global approach, but if the additional features of the hybrid approach are not needed, then we recommend that the global approach be used instead. The global approach results in a smaller code footprint as well as a reduced need for error checking by the global Koko instance resulting in a faster runtime.

3.5 Design Rationale for Koko

Koko seeks to facilitate socio-affective application development and to help developers focus on the innovative and creative aspects of those applications. To this end, Koko adopts the principles of service composition [45] to combine multiple services into a cohesive application. For instance, booST and hooPS (described in Section 5) combines mobile sensors from Android, the social network from OpenSocial, and the emotion modeling of Koko to create a cohesive experience geared toward increasing physical activity among youth (e.g., booST) or enhancing social interactions (e.g., hooPS).

Composing existing services enables developers to design lightweight applications quickly while not sacrificing features. We outline below the rationale for choosing the services and platforms that compose Koko and are subsequently used by both booST and hooPS.
Figure 3.7: Composition of the booST and hooPS socio-affective applications.

**Social networks.** booST and hooPS are designed around social interaction. By connecting to established social networks, Koko helps an application obtain relevant information and reach a potentially large user base. Accordingly, Koko connects to existing social networks using OpenSocial (a standard [35] that provides a unified interface for reading and manipulating contacts in major social networks including MySpace, Orkut, and Plaxo).

**Sensors.** Our ability to accurately model basic emotional states depends upon the sensors we can access. Modern mobile devices include many sensors [24], which benefit from near-constant presence. Koko can also access other sensors and services. For instance, hooPS uses a web service that provides real-time updates on college basketball games. Koko helps developers by supporting plugins for gathering and storing data for each sensor type; the plugins can be shared by developers.

**Affect model.** Creating and maintaining an affect model that predicts a user’s affective state is nontrivial for most developers, who are not specialists in affect modeling. Koko provides a configurable machine learning approach that predicts a user’s affective state based on data from the application and sensors. Thus, Koko facilitates developer productivity by
enabling them to focus on the creative and innovative aspects of their application.
Chapter 4

Methodology

The multiagent middleware Koko attempts to reduce the burden of incorporating affect modeling into applications. However, Koko can be effective only if the models it needs to function are suitably constructed based on information provided by the software developer. To this end, we have developed a methodology that employs expressive communicative acts as an organizing principle for affective applications. The methodology intuitively specifies the steps needed to create an affective application in Koko. Our key motivation behind this methodology is that it would facilitate the construction of an affective application by engineers who may lack a prior background in affective modeling.

Our initial attempt at such a methodology was called Koko-ASM [48]. Koko-ASM was a methodology for configuring a social (multiagent) affective application using Koko. Koko-ASM uses the concept of agent interaction and speech act theory [42] as a means of modeling communication between agents in a social environment. Using those models of communication, Koko-ASM, guided the application developer through the process of identifying the communicative actions among agents and then through a series of step decompose those interactions into the artifacts needed to configure Koko.

Using Koko-ASM we conducted several pilot studies with the goal of improving the methodologies process. For instance, as part of a course project, twelve undergraduate students (in
teams of two) implemented an socio-affective application using Koko-ASM. The teams were able to create prototypes within six weeks. The study was not used to evaluate the efficiency of Koko-ASM, but rather was used to find the weaknesses (e.g., lack of details) in the methodology so we could make improvements.

Now that we have laid a foundation by way of Koko and Koko-ASM we describe Kokomo, our new configuration methodology for Koko. The purpose of Kokomo is the same as that of Koko-ASM, to guide the application developer through the Koko configuration process in an intuitive manner. As such, two methodologies share many similarities, but they also have many critical and fundamental differences in how they achieve the goal.

One such difference is that Koko-ASM is designed for applications that have a high degree of interaction among Koko’s user agents. Koko however is designed to operate in environments that range from those that are highly social to those containing only a single agent. The Kokomo methodology is designed to work for all application types and as such removes Koko-ASM’s restriction that requires a interactive multiagent environment within Koko. Kokomo still uses the concept of agent interaction to guide the developer through the process. Though, in Kokomo we expand the definition of an agent beyond that of a Koko user agent. In Kokomo an agent can either be a Koko user agent (e.g., the user) or a virtual AI-controlled entity (e.g., virtual character).

Kokomo and Koko-ASM share many commonalities. For instance, both methodologies use speech act theory throughout the methodology. Additionally, some areas of Kokomo borrow some of concepts from Koko-ASM. However, no step in the methodology is equivalent as every step has been refined over the course of time. At the conclusion of each Koko-ASM study, we collected developer feedback on how to improve the methodology. The feedback has been incorporated into Kokomo in an effort to make the process as straightforward as possible for our target demographic. In fact, Kokomo adds two additional steps to provide more granular guidance for developers.

The remainder of this section elaborates on Kokomo. First, we describe the methodology
in detail in the next section. In the following section we provide an example of how the methodology could be used. Then in Sections 4.3 and 4.4 we empirically evaluate Kokomo’s utility through a developer study. The results are positive and demonstrate that the developers who used Kokomo were able to develop an affective application in less time, with fewer lines of code, and with a reduced perception of difficulty than developers who worked without Kokomo.

4.1 Description of Kokomo

Upon completion of the Kokomo methodology the developer will have identified or constructed all the artifacts needed to configure Koko for use in their application. Table 4.1 systematically lists the steps used to create an affective application with Koko. The documentation below highlights the key concepts of each step.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Artifacts Produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define the set of possible roles an agent may assume</td>
<td>Agent Roles</td>
</tr>
<tr>
<td>2</td>
<td>Describe the expressives exchanged between roles</td>
<td>Expressive Messages</td>
</tr>
<tr>
<td>3</td>
<td>Derive the emotions to be modeled from the expressives</td>
<td>Emotions</td>
</tr>
<tr>
<td>4</td>
<td>Describe a set of nonexpressive application events</td>
<td>Application Events</td>
</tr>
<tr>
<td>5</td>
<td>Construct the appropriate event definitions</td>
<td>Event Definitions</td>
</tr>
<tr>
<td>6</td>
<td>Select the sensors to be included in the model</td>
<td>Sensor Identifiers</td>
</tr>
<tr>
<td>7</td>
<td>Select the desired affect model from Koko’s repository</td>
<td>Model Identifier</td>
</tr>
<tr>
<td>8</td>
<td>Register with Koko’s runtime environment</td>
<td>Source Code</td>
</tr>
</tbody>
</table>

*Step 1* requires the developer to identify the set of roles an agent may assume in the context of the application. Remember, that in Kokomo an agent has an expanded definition when compared to Koko-ASM. Examples of such roles include teacher, student, coworker, enemy, and rival. A single agent can assume multiple roles and a role can be restricted to apply to the agent only if certain criteria are met. For example, the role of coworker may only apply if the two agents communicating work for the same company.
Step 2 requires the developer to describe the expressive messages or *expressives* exchanged between various roles [42]. Searle defines expressives as communicative acts that enable a speaker to express their attitudes and emotions towards a proposition. Examples include statements like “Congratulations on winning the prize!” where the attitude and emotion is congratulatory and the proposition is winning the prize. Formally, the structure of an expressive is

\[
\langle \text{sender}, \text{receiver}, \text{type}, \text{proposition} \rangle
\]

(4.1)

The type of the expressive refers to the attitude and emotion of the expressive and the proposition to its content, including the relevant events. The sender and receiver are selected from the set of roles defined in Step 1. The developer then formulates the expressives that can be exchanged among agents assuming those roles. In the case that both the selected roles can only be assumed by artificially intelligent agents then only formulate expressives for communicative acts that are observable by a user agent. The result is a set of all valid expressive messages allowed by the application.

Step 3 requires the developer to select a set of emotions to be modeled from Koko’s emotion ontology. The selected emotions are based on the expressives identified in the previous step. To compute the set of emotions, we evaluate each expressive involving a user agent and select the most relevant emotions from the ontology for that particular expressive. The selected emotions are then added to the set of emotions required by the application. This process is repeated for every expressive and the resulting emotion set is the output of this step.

Koko offers support for expressives because it provides a well-delineated representation for affect. Koko can thus exploit a natural match between expressives and affect to help designers operationalize the expressives they use in their applications. The recommended approach to selecting an emotion is to structure Elliott’s set of emotions [13] as a tree (Fig. 4.1). Each leaf of the tree represents two emotions, one that carries a positive connotation and the other a negative connotation. Given an expressive, you start at the top of the tree and using its type
and proposition you filter down through the appropriate branches until you are left with only the applicable emotions. For example, say that you have a message with a type of excited and a proposition equal to “I won the game.” Now using the tree you determine that winning the game is an action the user would have taken and that excited has a positive connotation, so the applicable emotion must therefore be pride. In general, the sender and receiver would have different interpretations. For example, if the recipient of the above message is the agent who lost the game, then the emotions that are relevant to the recipient would be admiration and reproach depending on their perception of the winner.

If the proposition of the expressive message is a composite or even ambiguous as to whether or not the type applies to an event, action, or object then more than one path of the tree may apply. Such is the case when an agent conveys its mood via an expressive message. Mood is an aggregation of emotions and therefore does not have a unique causal attribution. For example, an expressive might convey that a agent is generally happy or sad without being happy or sad at something. Therefore, we do not select any specific emotion when evaluating a expressive pertaining to mood as the emotions that comprise the mood are captured when evaluating
the other expressives. In other words, mood is not treated directly upon the reception of an expressive.

*Step 4* requires the developer to describe a set of non-expressive events in their application. A non-expressive application event occurs when the user has an interaction with the application that may effect their emotions. In particular we are interested in the emotions that were selected in the previous step. The non-expressive aspect of these events is that they are not a direct result of interaction between two agents, but rather the agent with their environment.

In theory, a developer could encode the entire state of the application as a set of events, but this is not necessary as we are only interested in the parts of the application that may effect the emotions we have selected. For example, the time the user has spent on a current task is likely to effect their emotional status, whereas the time until the application needs to clear its caches or garbage collect its data structures is likely irrelevant.

To help guide developers we encourage them to start by thinking about the interactions (direct and indirect) a user can have with the application and how those interactions relate to the set of emotions that were defined in Step 3. For example, if the application monitored phone calls and the user hadn’t sent or received a call all day then that may effect their emotional state. Further, after identifying an event it must be quantified. In the previous example the quantification could be the time since the last call was received.

*Step 5* requires the developer to construct the appropriate event definitions using Koko’s event ontology. The events described are a combination of the expressives in Step 2 and non-expressive application events identified in Step 4. Each expressive identified in Step 2 is modeled as two event definitions, one for sending and another for receipt. The decomposition of a message into two events is essential because we cannot make the assumption that the receiving agent will read the message immediately following its receipt and we must accommodate for its autonomy.

*Step 6* and *Step 7* both have trivial explanations. Koko maintains a listing of both the available sensors and affect models, which are accessible by their unique identifiers. The developer must simply select the appropriate sensor and affect model identifiers.
Step 8 completes the methodology by providing the developer with details on how to register their application within Koko’s runtime environment. Using the artifacts generated by the previous steps methodology we now have sufficient information to configure the Koko middleware. Given the artifacts gathered from the previous steps we can configure the application via the interfaces defined by Koko. Upon success the Koko registration interface returns an applicationID, which acts as the identifier for the application. The developer uses the applicationID in all subsequent interactions with the Koko runtime, such as sending information about the user to Koko or querying for the user’s current affective state. The formal description of the interaction can be seen in Equation 4.2. The outputs of Steps 1, 2, and 4 are not needed by this interface because they are encapsulated within the provided emotions and eventDefinitions.

\[
\text{emotions} \times \text{eventDefinitions} \times \text{sensorIDs} \times \text{modelID} \mapsto \text{applicationID} \tag{4.2}
\]

### 4.2 Example Usage

Using expressives to communicate an agent’s affective state extends traditional AOSE into the world of affective applications. We provide an illustration of that fact by conducting a case study that steps through the methodology and produces a functional affective social application. The subject of our case study is a social, physical health application with affective capabilities, called booST. To operate, booST requires a mobile phone running Google’s Android mobile operating system that is equipped with a GPS sensor.

The purpose of booST is to promote positive physical behavior in young adults by enhancing a social network with affective capabilities and interactive activities. As such, booST utilizes the Google OpenSocial platform to provide support for typical social functions such as maintaining a profile, managing a social circle, and sending and receiving messages. Where booST departs from traditional social applications is in the use of energy levels and emotional status.

Each user is assigned an energy level that is computed using simple heuristics from data retrieved from the GPS. Additionally, each user is assigned an emotional status generated from
the affect vectors retrieved from Koko. The emotional status is represented as a real number ranging from 1 (sad) to 10 (happy). A user’s energy level and emotional status are made available to both the user and members of the user’s social circle.

To promote positive physical behavior, booST supports interactive physical activities among members of a user’s social circle. The activities are classified as either competitive or cooperative. Both types of activities use the GPS to determine the user’s progress toward achieving the activities goal. The difference between a competitive activity and a cooperative activity is that in a competitive activity the user to first reach the goal is the winner, whereas in a cooperative activity both parties must reach the goal in order for them to win.

Koko hosts an agent for each booST user. A user’s agent maintains the affect model that is used to generate the user’s emotional status. booST provides the agent with data about its environment, which in this case is the user’s social interactions and their participation in the booST activities. The user agent processes the data and returns the appropriate emotional status. Further, Koko enables the exchange of affective state between booST agents (representing a social circle of users). This interaction can been seen in Figure 5.1 in the emoticons next to the name of a buddy. The affective data shared among members of a social circle is also used to provide additional information to the affect model. For instance, if all the members of a user’s social circle are sad then their state will have an effect on the user’s emotional status.
booST Configuration

Table 4.1 outlines the process for creating an affective, social application. We now demonstrate this process using booST.

Step 1 requires that we identify the set of roles an agent may assume. booST involves only two roles, friend and self. An agent A assumes the role of agent B’s friend if and only if the users represented by agents A and B are members of each other’s social circle. The social circle is maintained by booST and can be equated to the friend list in popular social applications such as Facebook and MySpace.

Step 2 requires that we identify and describe all expressives that occur between the two roles. Below is an example of what a few such messages would look like depending on the outcome of a competitive activity within booST. The remaining messages would be defined in a similar fashion.

Step 3 requires that we select a set of emotions to model from the emotion ontology. As Section 3.1 shows, the ontology is based on Elliot’s expansion of the OCC model, which categorizes emotions based on the user’s reaction to an action, event, or object. When inspecting each expressive, we find that booST focuses on measuring happiness and sadness of the user with respect to actions and events. Therefore, we can narrow our selection to only emotions that meet those criteria. As a result we select four emotions: two are focused on the actions of the user (pride and shame) and two on the events (joy and distress). booST uses these emotions to compute the user’s emotional status by correlating (1) pride and joy with happiness and (2) shame and distress with sadness.

Step 4 requires that we describe the nonexpressive application events using the event ontology. Since an event in booST is merely an instantiation of the event ontology, the event descriptions are trivial. For example, a nonexpressive event could be whether or not the user is logged into game. This simply translates to an ordinal value in the ontology with the two states being on or off. Given the social nature of booST we may wish to share this information between agents. In that case, we specify the event as public, which allows other agents in the
user’s social circle assess to that information.

Step 5 requires that we construct the event definitions using the event ontology. The complete set is formed using the events from Step 4 and the expressive messages from Step 2. Each expressive message yields two events: a sending event and a receiving event. The resulting events provide Koko with the necessary details about the application’s environment (Table 4.2 shows some examples). Since an event in booST is merely an instantiation of the event ontology, the event descriptions are trivial. For example, the “Competitive Exercise Challenge” message can be described as an action that involves another agent. When an event occurs at runtime, the context associated with its occurrence would contain attributes such as the time of day, challenge information, and the user’s energy level as calculated by the application. The context is treated as a special event and like other events it’s attributes are defined using the ontologies primitives.

<table>
<thead>
<tr>
<th>#</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>View my energy level and emotional state</td>
</tr>
<tr>
<td>2</td>
<td>View my friend’s energy level and emotional state</td>
</tr>
<tr>
<td>3</td>
<td>Send or receive “Cooperative Exercise Challenge” message</td>
</tr>
<tr>
<td>4</td>
<td>Send or receive “Competitive Exercise Challenge” message</td>
</tr>
<tr>
<td>5</td>
<td>Complete or fail a cooperative activity</td>
</tr>
<tr>
<td>6</td>
<td>Complete or fail a competitive activity</td>
</tr>
</tbody>
</table>

Step 6 requires that we select the sensors to be included in the model. As we have already noted booST requires a single sensor: a GPS. The first time a sensor is used a plugin must be created and registered with Koko. The GPS plugin converts latitude and longitude vectors into distance covered over a specified time period. This data is then maintained by Koko and made available for consumption by both the application and the affect model.

Step 7 of the process is trivial as booST employs an affect model that is provided by Koko. In particular, booST employs the model that implements decision trees as its underlying data
structure.

**Step 8** completes the process by registering booST with the Koko runtime environment. Using the artifacts generated by the methodology we now have sufficient information to configure the Koko middleware. Following Equation 4.2 the required artifacts are supplied to the runtime resulting in an application identifier that is unique to booST. Further, upon configuration Koko maintains an agent for each booST user. The agent is responsible for modeling the affective state of the user as well as communicating that state to other agents within the user’s social circle. With Koko supporting the affective aspects of booST, application developers are free to focus on other aspects of the application. For instance, they may focus on developing the creative aspects of the game, such as how to alter gameplay based on the user’s affective state.

### 4.3 Developer Study

We conducted a developer study to determine the actual and perceived benefits of Koko and Kokomo to application developers. Our hypothesis is simple: developers who use Koko and Kokomo can more easily construct an affective application than those who do not, while at the same time not diminishing the quality of their application. The study consisted of the same application assigned to groups of developers employing Kokomo (with Koko), Koko alone, and neither (just traditional techniques). The remainder of this section focuses on the application used in the study and the following section provides an empirical analysis of the results.

By its definition every ‘developer study’ involves the development of software in some fashion. In this case, our study is focused around the development of a mobile application, in particular a math tutoring application for high school students. The architecture of the application centers around a dynamic lesson planner which we call the virtual teaching assistant (vTA). The goal of the vTA is to sharpen the user’s mathematical skills in a variety of areas, such as probability and geometry. The target audience is students preparing for standardized tests, in particular the standardized end-of-grade tests that are given to American high school students at the conclusion of the 8th and 12th grades.
Our choice of this particular application was deliberate in that it satisfied three specific criteria. First, we choose the application because it met the time and complexity constraints of our study. The application itself is simple enough to implement in a short period of time, but it is also complex and open enough that each developer’s final application could be significantly different from that of their peers. Second, as you will see in the following section, the application can be compartmentalized into discrete units which allows us to evaluate individual components of the application across all developers. Third, and perhaps most importantly the application domain was one that lent itself to affective interactions. For example, the role of affect in the learning process is an area with well documented psychological foundations [41] and is an active research in computer science [25]. The combination of those three criteria in the chosen application provide sufficient basis by which we can evaluate our hypothesis.

![Flow Channel](image)

**Figure 4.3: Flow Channel.**

Before proceeding to outline the details of the application we first outline the psychological foundation that explains the role of affect in the learning process. The primary motivation comes from the theory of flow that was developed by Mihály Csíkszentmihályi [8]. A high level simplification of the theory is that individuals learn best in situations where they are neither extremely bored nor frustrated, but instead at a median between the two. The median or sweet
spot between boredom and anxiety was given the term "flow channel" (see Figure 4.3). This theory has been evaluated by Csíkszentmihályi [9] on student concentration, but the concept has also been applied to research in virtual learning environments [33].

4.3.1 Application Details

When designing and developing software it is often helpful to conceptualize the software as a set of components with defined interactions as we have done in Figure 4.4. At all times the user interface is under the control of either the virtual teaching assistant (vTA) or the testing algorithm. The remaining boxes represent the various services that are available to be used by either the vTA or testing algorithm.

![Figure 4.4: Application Components.](image)

Prior to describing the details of each individual component we must first explain how they interact. The expected application usage is as follows. Initially, when the application is opened the user is greeted with a welcome screen at which point they can elect to start a tutoring session. A tutoring session is comprised of 15 different mathematical problems where the first 10 problems are selected by the vTA and the remaining 5 by a testing algorithm. The goal of the vTA is to select the 10 best training problems in order to prepare the user for the test. The
vTA has at its disposal a database of all possible questions as well as a set of physical sensors that can provide feedback on the user’s current state. Using those services and by collecting other data such as the user’s history, environment, and current affective state the vTA crafts a customized lesson plan for each user. Using the customized plan the application presents the user with a series of problems (defined by the plan) and is then tested via the testing algorithm as a means of evaluating the effectiveness of the vTA’s plan.

The developer is not responsible for all aspects of the application. In fact, each developer is provided working code for all components of the application, with one exception, we omit the vTA. Using the theory of flow as motivation the developers are responsible for designing and implementing their own customized vTA. The choice to have them only implement the vTA was carefully planned in order to strengthen the results of our study. In this manner, we reduce non-affect related variability in the solutions and our measurements of quality and effort, and force our developers to focus on the logic of the application.

Now that we know the expected application usage and that the developer’s focus is on the development of the vTA we look at each component of the application. Each component is described from the developer’s perspective. Some components will be tools used by the vTA in an effort to meet its goal while others will serve as an outlet for information produced by the vTA.

Problem Set (Question Database)

The problem set is a subset of a question bank that is maintained by the National Center for Educational Statistics (http://nces.ed.gov). The questions are extracted from data collected by the National Assessment of Educational Progress. In particular, we extracted all multiple choice mathematics questions asked to 8th through 12th grade students.

Each problem in the dataset contains an id, question, grade level, content area, difficulty, and answer. The questions are text based with the occasional greyscale illustration and included 5 multiple choice options as answers. The grade level specifies the grade at which this question was
targeted. The content area defines the problem type as being either properties and operations, geometry, analysis and probability, or algebra. Finally, the difficulty and answer are linked in that the one is the correct answer for the problem and the other is the percentage of students who correctly answered the question on national standardized tests.

The questions are exposed to the developer using a small set of interfaces. These interfaces allow the developer to retrieve a set of questions based on any permutation of the problem components. The retrieval mechanism is analogous to standard SQL queries that most developers are familiar with. Additionally, as a convenience developers can also request a randomly selected problem that meets a set of specified criteria. For example, they can request a random geometry problem that less than 20% of students answered correctly.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Unique identifier for each question</td>
</tr>
<tr>
<td>Question</td>
<td>A single question consisting of text and on occasion a grayscale illustration</td>
</tr>
<tr>
<td>Answer</td>
<td>The answer to the question (A - E for multiple choice)</td>
</tr>
<tr>
<td>Grade</td>
<td>Grade level this question was targeting (8th - 12th grade)</td>
</tr>
<tr>
<td>Content Area</td>
<td>Either Properties and operations, geometry, analysis and probability, or algebra</td>
</tr>
<tr>
<td>Difficulty</td>
<td>Percentage who correctly answered the question on national standardized tests</td>
</tr>
</tbody>
</table>

**User Interface**

As shown in Figure 4.4, the user interface is controlled by the vTA. The majority of the interface is provided to the developer and the developer controls the UI via a set of predefined control points. These predefined points allow the developer basic control of the UI without having them actually design and control the entire user experience. The user interface is comprised of 4 screens with the two primary screens shown in Figure 4.5. Using these basic screens the vTA is allowed to configure the user experience with the following four actions:

**Pre-Session Questionnaire.** Just before a training session begins, the vTA may present the
user with a set of up to three questions, and use the answers received to initialize the lesson plan for the user.

**Ask Question.** After initialization the UI requests a question from the vTA, who responds with the ID of a question, which the UI displays on the screen.

**Record Result.** The UI conveys the user’s answer to the vTA. The UI then transitions to the screen that shows the user the correct answer.

**Post-Question Feedback.** After each question, the vTA can optionally ask the user one multiple choice affect-related question, which can provide the vTA with better insight into the user’s current affective state.

![Image](image_url)

(a) Question Presentation  (b) Result and Feedback

Figure 4.5: The application’s user interface.

The vTA controls the user interface for the first 10 problems to train the user, at which point the testing algorithm assumes control and asks the final 5 questions. The testing algorithm can utilize the same points of interaction as the vTA, but it also has the added ability to display the most recent user’s history on the application’s start page.
Mobile Sensors

A unique advantage of building a mobile application is the availability of sensory inputs. Most smart-phones on the market today come equipped with sensors like GPS, accelerometer, microphone, and proximity sensors. Access to data provided by these sensors is useful for determining the environment in which the user is operating. Using this data developers can track the performance of users in specific environments and then customize the lesson plan accordingly. Additionally, environmental data is also beneficial when evaluating the emotional state of the user.

The Android operating system was chosen as the development platform and each developer had access to a development device in order to test their application with real sensor inputs. The operating system has utility classes for a variety of sensors that provide easy to use interfaces for extracting data from the sensors.

Virtual TA

The virtual TA is at its core an intelligent agent that maintains a reasoning engine. It is in this engine where the developer must use the available tools, such as sensors and the question database, to craft a unique lesson plan for the user. The vTA has one simple goal create the lesson plan for the user that gives them the best chance of success when tested.

In general, a vTA could carry out complex reasoning, such as based on the user’s previous standardized test scores and available national statistics. However, allowing such extensive approaches would have introduced too many variables into the evaluation. To reduce the variability, we instructed all developers to consider only three categories of information (exam, environmental, and emotional data) in their reasoning algorithms to formulate a lesson plan. Using data gathered from those categories the developer must write a reasoning algorithm that formulates the lesson plan for the user. All lesson plans are considered to be dynamic and can be changed after the user answers each question.

Exam data is considered to be any data that can be computed the user’s past performance.
Looking at the user’s performance is often good indicator of what types of problems they do well on and where they struggle. It is up to the developer how complex the analysis of the exam data is, but they must at least consider the previous question and the overall performance of the user on the previous exam.

Environmental data is considered to be any data that is gathered from a physical sensor. The developer was instructed to use at least one sensor (see Section 4.3.1) available on the mobile device. For instance, by using the microphone on the device a developer could monitor how well the user performed in noisy environments. The developers must integrate data from at least one sensor into the reasoning logic of their vTA.

Emotional data is qualitative feedback from the user on their current status. At the conclusion of each question provided by the vTA they can gather multiple choice feedback from the user. This feedback is to be used to determine how the user felt about the last question they were asked. The developer are then to use this feedback combined with Csikszentmihalyi’s theory of flow as a means of updating the lesson plan. Developers are required at a minimum to use this data to influence the next question to be presented as well as store the feedback for future analysis when selecting questions.

Using data from the specified categories the developer must finally construct a reasoning algorithm that uses the data to generate a lesson plan. In addition to writing the algorithm each developer was required to provide written documentation detailing how the algorithm functioned.

Testing Algorithm

The testing algorithm is responsible for selecting the five questions that will be used to evaluate the user at the end of each training session. The selection of the problems is not entirely random, but instead ensures that the user is tested over a broad set of problems.

The algorithm considers the question category, difficulty, and recurrence. The algorithm ensures that no more than two questions are asked from the same category. Therefore, since
there are only four categories the test always covers a minimum of three categories. The algorithm also ensures that the national average for correctly answering at least one question is above 70% correct (easy) and one is below 30% (hard). Finally, the algorithm ensures that after a question is asked it will not appear again for the next three exams. All questions meeting this criteria are put into a set from which one question is selected and the process is repeated.

The details of this algorithm was given to the developers so that they were aware of the testing strategy. Further, the software component implementing the algorithm stores the user’s past performance on exams. This data can be accessed by the vTA and is also used to display the five most recent testing results to the user.

4.4 Empirical Evaluation

Our evaluation measured the ease of constructing an application both subjectively and objectively. We collected subjective developer feedback on their perceived difficulty via surveys which we obtained throughout the duration of the study. The feedback indicates a lower perception of difficulty for the affective portions of the assignment when using Kokomo than without it. Analysis of objective difficulty measures (code metrics and effort analysis) shows that Kokomo yields the best results on nearly every measure of code complexity and effort. However, the results were not uniformly strong for developers employing Koko alone and we revisit this point in Section 6.

We selected 30 students to participate in the developer study. Each developer had experience programming in Java, and have no prior experience in affective computing. The developers were paired into teams of two and given basic programming assignments in order to evaluate their programming abilities. The assignments were evaluated by a third-party reviewer who ranked each team based on its programming proficiency.

Additionally, each developer was given a survey asking them to rate their software engineering experience and describe projects they had worked on. Over 97% of the developers had a minimum of two years experience programming in Java, but for 88% of developers the initial
programming assignments were their first exposure to programming in the Android operating system.

Using these results of the programming assignments and surveys we ranked each individual developer with respect to their peers. Using those rankings, we then divided the teams into three groups so as to equalize the estimated programming ability across the groups. Each team within a group was assigned a variation of the same task. The control group did not have access to Koko or Kokomo, one of experimental groups had access to Koko, and the second experimental groups had access to Koko and Kokomo.

All groups are given identical instructions for the application. The Koko and Kokomo groups were also provided instructions for using Koko and Kokomo, respectively. Each team was given four weeks in which to complete the assignment. They had to design and develop the vTA agent using either the components provided (e.g., question database and user interface) or any API provided by the Android SDK.

We conducted a baseline quality check for each application. This check ensured that all applications ran appropriately and did not omit any of the features in the required feature set. This test was done in order to strengthen our claim that the introduction of Koko and Kokomo does not diminish the quality of the application. All applications from all three groups passed the baseline check. In fact, we observed that applications using Koko and Kokomo had additional features, in particular, affective features that were not present in the control groups applications. However, our statistical measures disregard such extra features.

Each team was evaluated in the same manner regardless of its group. We evaluated the study in both objective and subjective ways. The objective portion involved measuring the time spent by each developer and various measurements on the source code. The subjective portion was based on periodic surveys of the developers to measure their perceptions regarding their feelings on the complexity of each aspect of the project and the utility of the tools provided. The complete surveys can be found in Appendix A.
The in-study surveys were completed by each developer every time they worked on the project. A minimum of one survey was required for each day (and within one day) the developer worked on the project. The survey collected information such as the time spent on each component of the application as well as their perceptions and comments on difficulty.

The post-study survey was completed at the end of the development cycle to evaluate the developer’s perception of the entire project. Additionally, this survey was used as a mechanism for developers to recommend improvements to the project and the tools.

The remainder of this section focuses on the evaluation of our hypothesis. Our hypothesis stated that developers who use Kokomo can more easily construct affective applications than those who do not, while at the same time not diminishing the quality of their application. The following two sections evaluate the objective and subjective metrics respectively. Finally, after viewing each category of metrics independently we take a holistic view of the results to determine if our claim is satisfied.

4.4.1 Objective Results

Measuring the complexity of a project based on the resulting source code is nontrivial because there are no definitive measures of complexity. Complexity can relate to the size of the code base and also to less measurable notions such as extensibility and maintainability. Here we adopt some well-known metrics from software engineering literature [16, ?], and use them to analyze each project independently. Finally, we compare the metrics across all three developer groups. Figure 4.6 highlights the results for four of the most well-known metrics. In all cases except Figure 4.6(d), the lower value indicates a better result.

**CC** McCabe’s Cyclomatic complexity [31] indicates the number of “linear” segments in a method (i.e., sections of code with no branches) and therefore helps determine the number of tests required to obtain complete coverage. It also indicates the psychological complexity of a method.
(a) Cyclomatic Complexity
(b) Number of Levels
(c) Number of Statements
(d) Number of Methods

Figure 4.6: Software metrics.
**NoLm** The number of levels per method reflects the number of logical branches each method has on average. NoLm is a key factor in determining code readability, and is also used to determine how well the code adheres to object-oriented design patterns.

**NoS** The number of statements in the project is a common measure of the amount of time spent developing and the general maintainability of the code.

**NoM** The number of methods in the project reflects increasing modularity and readability of the code (for a fixed NoS value).

We had the developers log the time they spent on each portion of the assignment. Figure 4.7 gives an overview of the time spent on the project as a whole as well as its affective components specifically. It is important to note that though the total project time did not drastically decrease with Kokomo, the percentage of time spent on affect fell sharply. Additionally, the drastic reduction of time and minimal variance for Kokomo in Figure 4.7(c) illustrate the benefit of the methodology.

### 4.4.2 Subjective Results

Several survey questions asked the developer to either rank components against one another or rate the individual components on a difficulty scale. Table 4.4 shows the average difficulty rankings given by each development group for the five key components of the project. You can see that the introduction of both Koko and Kokomo reduced the developer’s perception of difficulty for the affective aspects of the project.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Control</th>
<th>Koko</th>
<th>Kokomo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Easiest)</td>
<td>Heuristic</td>
<td>Heuristic</td>
<td>Affect</td>
</tr>
<tr>
<td>2</td>
<td>Programming</td>
<td>Affect</td>
<td>Heuristic</td>
</tr>
<tr>
<td>3</td>
<td>Exam</td>
<td>Programming</td>
<td>Environment</td>
</tr>
<tr>
<td>4</td>
<td>Affect</td>
<td>Environment</td>
<td>Programming</td>
</tr>
<tr>
<td>5 (Hardest)</td>
<td>Environment</td>
<td>Exam</td>
<td>Exam</td>
</tr>
</tbody>
</table>
Figure 4.7: Development time metrics.
Further, we asked developers to individually rate each component of the assignment on a difficulty scale of 1 (least) to 5 (most). As shown in Figure 4.8(a), 90% of the developers using Kokomo rated the difficulty of the affective component as a 1 or 2. However, 70% of the control development group gave the same component a rating of 3 or higher. Finally, Figure 4.8(b) shows how developers rated the benefit obtained from the Koko tools on a benefit scale of 1 (none) to 5 (high). Of the developers using Kokomo, 70% perceived Koko to offer a high benefit to them (a rating of 4 or 5) whereas only 40% of those without Kokomo gave a similar rating.

![Figure 4.8](image)

**Figure 4.8:** Subjective Developer Distributions.

### 4.4.3 Conclusions

Using both objective and subjective data, we have empirically shown that the introduction of Kokomo results in both perceived and tangible benefits for developers. With respect to software metrics, the applications written using Kokomo had either equivalent or better results than those written without Kokomo. The use of Kokomo resulted in significant time savings and also significantly reduced the time variance for constructing the affective component. The reduction of the time variance is important in that it enables developers to more accurately budget their time. More accurate time estimates in turn allow them to better scope the cost of
incorporating affect into an application.

Further, in all cases where developers were asked their perceptions of the project those using Kokomo had a more positive perception than those who did not. This positive perception of Kokomo combined with the tangible benefits make a compelling case for practical use by software developers.
Chapter 5

Socio-Affective Applications

We now describe two socio-affective, youth-centric applications that serve the design goals we discussed in the Introduction. These applications demonstrate two main technical modes of connectivity, the asynchronous and the synchronous, respectively. These lead to somewhat different user experiences, but demonstrate connectivity in general.

5.1  booST: Promoting Health

The motivation behind booST is to boost the user’s “energy” via physically exercise encouraged through the personal pressure and exhortations of friends. booST is designed as an extension to the user’s existing social network and seeks to facilitate interactive physical activities among members of a user’s social circle. An example of a physical activity supported by booST is running a specified distance within an allotted time.

booST classifies its (goal-directed, measurable) activities as either competitive or cooperative. In a competitive activity, the user to first reach the goal is declared the “winner,” whereas in a cooperative activity both parties must reach the goal in order for them both to win equally.

To participate in an activity, a user issues a challenge (see Figure 5.1b) to a friend. The friend thus challenged can either accept or decline the challenge. If the friend accepts the challenge, both participants must complete the challenge activity in the allotted time. Importantly,
they can proceed independently of one another, indicating the asynchrony of *booST*. One of the participants may complete his jog around the park in the morning; the other may run after sunset in his neighborhood. Asynchrony enables each participant to follow a personal schedule within the time interval specified in the challenge.

![Figure 5.1: *booST* screenshots. (1) The home screen shows the user’s current status as well as a list of friends who also subscribe to *booST*. (2) The challenge screen where a user describes the conditions of the challenge he throws at a friend. (3) A screen showing a challenge that has been accepted, but has not yet been completed.](image-url)

One of the unique features of *booST* is that it tracks two special application-specific metrics for each user, the *energy level* and *emotional status* of each user. A user energy level is computed using simple heuristics based on data from the application and from available sensors. The energy level rewards users for participating in challenges, winning activities, and for their general physical activity. A user’s emotional status represents the user’s current happiness level. The emotional status is represented as a number ranging from 1 (unhappiest) to 10 (happiest). *booST* makes a user’s energy level and emotional status available both to the user and to members of the user’s social circle.

The use of physical activity in *booST* helps keep the users grounded in their physical environment. Similarly, the inclusion of emotions helps the user’s social interactions more
personal. For instance, because a user’s friends can access his or her emotional status, they can encourage the user through various interactions. Further, *booST* incentivizes users to encourage any friends with a low emotional status value by giving them large energy boosts for participating in cooperative activities with such friends.

We piloted *booST* with a small number of users to better understand their usage patterns. In a typical usage, users were more likely to challenge their real-world friends, not just those on an online social network. Further, we observed that users who exercised regularly prior to using *booST* simply incorporated the challenges into their daily exercise patterns. When issuing their own challenges, such “energetic” users would issue a challenge to a less energetic friend in order to earn the biggest gain to their energy from within *booST*. This tended to encourage the less energetic users, who prior to using *booST* did not exercise daily, to complete at least some of the challenges sent.

### 5.2 *hooPS*: Strengthening Friendships

The *hooPS* application is centered around socially connecting users with their friends while they simultaneously watch a basketball game. The motivation behind *hooPS* is to turn what might otherwise be an isolating and passive activity into an engaging episode.

Using *hooPS*, users select a game to watch and can chat concurrently with friends watching the same game. *hooPS* monitors a user’s basic emotional state and shares it with friends present within the current *hooPS* session. Based on a continual assessment of the user’s—and the user’s friends’—emotional state, *hooPS* recommends messages to send. For example, as the user’s favored team’s prospects ebb and flow during the game, the user’s emotional state would change as well.

Unlike *booST*, *hooPS* is conceptually synchronous. Users interact with one another in real-time while watching the same game. In addition to enabling user to exchange messages about the game, *hooPS* provides information such as the score, time remaining, and other pertinent game related information.
Figure 5.2: *hooPS* screenshots from a student’s implementation. (1) The startup screen that displays the currently available basketball games. (2) The game screen that displays the status of the game as well as the messages exchanged between friends. (3) The message recommendation interface that provides suggestions based on both the sender and receiver’s emotional state.

We piloted *hooPS* with a small number of users. In a typical usage, we found that user’s were willing to interact with users in their social network that they did not maintain a consistent relationship with in the physical world. Users overwhelmingly reported that they felt connected to the other participants through the basketball game. In fact, users conversed nearly equally between fans of their team and those of their rivals.
Chapter 6

Discussion and Literature

It is worth including some additional remarks about our efforts and the work of others. If affective computing is to have the practical impact that many hope it will, advances in software architecture and methodology are crucial. This chapter serves as a way to address issues that have been raised as a result of these advances both in our work and the work of our peers. Further, we have broken the chapter topically in an effort to coherently addresses a variety of issues across all areas of our research. Furthermore, at the conclusion of this section we elaborate on the future directions of our research.

Affect Modeling. Existing appraisal theory applications are developed in a monolithic manner [13] that tightly couples application and model. As a notable exception, EMA provides a domain-independent framework that separates the model from the application. Whereas EMA focuses on modeling virtual characters in a specific application, Koko models human emotion in a manner that can cross application boundaries.

We adopt appraisal theory due to the growing number of applications developed using that theory. Our approach can also be applied to other theories such as Affective Dimensions [37], whose models have inputs and outputs similar to those of an appraisal model. Likewise, we have adopted Elliot’s set of emotions because of its pervasiveness throughout the affective research community. Its selection does not signify that Koko is bound to any particular emotion ontology.
Therefore, as the field of affective computing progresses and more well-suited ontologies are developed, they too can be incorporated into the architecture.

The CARE affect model that Koko supports by default is not the only model of its kind. CARE falls in the class of Cognitive-Based Affective User Modeling (CB-AUM) [30]. For example, Conati developed a CB-AUM style model using dynamic Bayesian networks as a means to model student emotions in pedagogical games [7]. We hope to include additional CB-AUM style models, which will give affective developers additional choices when constructing their affective games.

Further, CB-AUM is not the only promising modeling approach when it comes to affect recognition using appraisal theory. For example, consider the PAD-based multimodal fusion approach employed by Gilroy et al.[18]. The fundamental difference between PAD-based fusion and CB-AUM is that CB-AUM models produce a vector of discrete affective states, while the fusion models produce a coordinate in a continuous three-dimensional affective space. The multimodal fusion approach also evaluates each mode of input (e.g., speech recognition or gesture recognition) independently and then fuses the results together, whereas typical CB-AUM models consider all modes of input using a single algorithm. Both approaches have their merits and depending on the application domain one may be more appropriate than the other.

Supporting multiple model types would make the architecture more appealing to game developers, by providing additional models that may better apply to their application domain. To achieve this, we could design another version of Koko that uses PAD-based multimodal fusion models instead of CB-AUM, thereby enabling applications of that type to benefit from the features offered by Koko. However, a more interesting challenge would be to support both model types in the same runtime environment and thus enable agents to communicate regardless of the underlying implementation. Accomplishing this requires a redefinition both of Koko’s interagent communication and its emotion ontology as well as a mapping function to translate between model output formats or to some newly defined format.
Virtual Agents. Koko’s interagent communication was developed with a focus on human-to-human social interactions (for example, booST). This does not limit Koko to only those interactions and we have begun to explore the usage of Koko with human-to-virtual agent interactions. Given the correct permissions, virtual agents (operating outside of Koko) could request the user’s affective state in the same manner as agents internal to Koko can. For example, the virtual agent models used in the ORIENT [28] and NonKin Village [44] applications could access the affect state of the player and use that information to enhance the agent’s decision making process.

Sensors. The future of physical sensors in games is promising. In the past decade, we have seen the popularity of motion-sensitive game controllers. Further, we have seen the emergence of gaming on mobile devices many of which employ sensory input. For instance, geolocative games have found their way into nearly every mobile platform that is equipped with GPS. Koko is poised to take advantage of new sensor technologies as they emerge and become widespread, thereby providing Koko’s affect models with an increasingly complete picture of the user’s environment.

Koko’s sensor interfaces are designed to require minimal processing. This design preference was motivated by a wish to ensure minimal latency between a change in the user’s environment and the time at which the agent is aware of the change. However, there have been recent advances in realtime processing of environmental data as a means of determining affective state, such as Wagner’s smart sensor integration (SSI) framework [56]. The SSI framework supports realtime affect recognition from sensors and can output the resulting affective state in a variety of formats including a format similar to our emotion ontology. The integration of SSI into Koko would enable Koko to reduce the number of attributes the primary model must learn by offloading that work to the sensor.

Game Integration. Due to the social and multiplayer nature of Koko, it cannot be contained within a traditional gaming engine. However, Koko can interoperate with gaming engines in
a loosely coupled manner. To incorporate Koko into an existing game engine API, the engine can simply provide a façade (wrapper) around the Koko API. The façade is responsible for maintaining a connection to the Koko service and marshalling or unmarshalling objects from the engine’s data structures to those supported by the Koko. Currently, Koko has service endpoints that support the communication of data structures encoded as Java objects, XML documents, and JSON objects.

**Existing Methodologies.** An important contribution of Koko and Kokomo is the incorporation of expressive communicative acts. These acts, though well-known in the philosophy of language, are a novelty both in agent-oriented software engineering and in virtual agent systems. The incorporation of expressives allow agents to interpret the difference between an expression of feelings and a statement of fact, thus enabling agents to better model their environment.

Existing AOSE methodologies specify messages at a high level and therefore are not granular enough to support expressives [11]. Kokomo’s applicability has a much narrower scope than these methodologies since it is restricted to applications that are both affective and social. These distinctions are simply the result of a difference in focus. It is quite possible, given the narrow scope of Kokomo, that it could be integrated with broader methodologies in order to leverage their existing processes and tools. For example, many methodologies have detailed processes by which they help developers identify all possible messages that are exchanged among agents. Kokomo would benefit by integrating those processes, thereby making it easier to identify the expressive messages.

**Developer Study Results.** The most perplexing result of our study was that the data showed that the introduction of Koko without Kokomo did not result in a marked improvement. Although this result does not invalidate our hypothesis, the fact that it appeared at all was initially surprising. Upon further evaluation of the code and the developer surveys, we did find an explanation for it. The developers who were only given Koko frequently recorded in their surveys that they were not able to completely grasp the affect modeling concepts that underlie
Koko. As a result, they were unsure of how to configure the Koko middleware. Whereas the above observations strengthen the case for Kokomo’s usefulness, we plan to spend time improving our documentation regarding Koko for future studies and deployment.

Koko is a multiagent middleware supporting affective interaction among agents, but this study involved only a single agent. This choice was intentional in that it enabled us to demonstrate that unlike Koko-ASM, the Kokomo methodology could be used in single agent environments. In the future, we plan to demonstrate the utility of Kokomo in multiagent settings by using it in social applications that take full advantage of Koko’s social design.

Further, we also found when inspecting the code that the affect modeling complexity in applications without Koko was minimal. The majority of non-Koko applications only met the baseline requirements for their affect model and none were as complex or adaptive as the models provided by Koko. These results have served as preliminary findings that merits further investigation. In particular, it would be valuable to find the point at which an application’s affect model is sufficiently complex to warrant the overhead of using Koko.

**Affective Gaming.** Commercial games, such as the popular series Mass Effect [5] and Assassin’s Creed [53], put a focus on engaging players emotionally through both gameplay techniques and interaction with virtual agents. It is the goal of Koko to enable developers of these types of games to enhance such emotional engagement between the game and the player by giving the developer insight into the emotional state of the game’s users. We have focused our attention on research games, such as Treasure Hunt [32] and Prime Climb [6] primarily because we lack visibility into the modeling techniques employed in commercial games.

**Enhanced Social Networking.** Human interactions rely upon social intelligence [19]. Social intelligence keys not only on words written or spoken, but also on emotional cues provided by the sender. Koko provides a means to build social applications that can naturally convey such emotional cues, which existing online social network tools mostly disregard. For example, an advanced version of booST could use affective data to create an avatar of the sender and
have that avatar exhibit emotions consistent with the sender's affective state. An important contribution of Koko is incorporating expressive communicative acts, which although well-known in the philosophy of language are a relative novelty both in agent-oriented software engineering and in virtual agent systems.

**Socio-Affective Applications.** It is worth including some historical remarks about our effort. We began with the concrete case of booST as a way to show how we could use mobile technologies to promote healthful behaviors among youth. As we began to conceptualize booST, we recognized the importance of incorporating a model for affect. Next, we realized that the programming effort required to build social-affective applications is substantial. This led to us abstracting out many key functionalities for dealing not only with affect, but also with (ambient) sensors and social networks into the Koko middleware. A further benefit is that a social application is potentially viral in the sense of users encouraging other users to adopt it.

Because we assume limited hardware as our baseline, the physical activities that booST can recognize must involve a change in position. If we have access to a sensor such as an accelerometer, we could incorporate an activity based on the number of hops or steps a user takes; if we have access to a sensor such as a heart-rate monitor, we could incorporate an activity such as maintaining a certain cardiovascular pattern for a specified duration.

Indeed, when we first conceptualized booST, it was to use a rich set of sensors (including for skin conductivity, heart rate, and blood pressure) as supported by the industry group Continua Health Alliance (http://www.continuaalliance.org/). Later, we decided to focus on readily available hardware which we could more easily hope to deploy.

We don’t claim that we have found the *killer application* of socio-affective mobile technologies. For that matter, we would claim that no such application exists. And, when dealing with connected youth, who are perhaps the most fickle demographic there is, it would be futile to try to find such an application. Instead, by providing a middleware that supports interesting concepts and improves developer productivity, we can facilitate the development of a large number of applications. Each would be a niche application for a particular group of youth with
a particular social context and preferences. It is its potential set of applications that makes Koko a valuable contribution to pervasive computing.

6.1 Future Directions

Koko and Kokomo open up promising areas for future research. In particular we focus on the architecture, methodology, socio-affective applications, and formalization of expressives. In the remainder or this section we briefly discuss each area.

**Enhanced Architecture.** The architecture is the centerpiece because in some fashion both the methodology and formalization efforts depend on the architecture. Currently Koko only supports a limited number of affect models all of which conform to the CB-AUM modeling style. An expansion in the the number of supported models makes Koko more viable for prospective developers. Additionally, there have been recent advances in real-time sensor architectures [56] that would greatly enhance Koko’s sensor infrastructure.

**Methodology.** The Kokomo methodology has been empirically proven to benefit developer in a single agent environment. However, only Kokomo’s predecessor Koko-ASM was evaluated in a multiagent scenario. Therefore to be thorough we also need to evaluate Kokomo in a multiagent environment.

**Socio-Affective Applications.** An implementation of a socio-affective application exists in the form of booST. To evaluate the merits of using affect within a socio-affective applications such as booST, we would like to conduct a user study. The hypothesis of this study is that socio-affective applications, in particular booST, are effective social motivators. In the case of booST we will evaluate it’s effectiveness at encouraging exercise and building stronger social bonds between participants.
**Expressives.** Finally, we see value in formalizing expressives as a medium for affective communication between agents. Currently no such formalization exists resulting in a gap between human communication and what agents can represent. The definition of expressives provided in this paper (see Section 2.2) gives us a basis for the formalization. Our future work must expound on that basis to ensure we are providing an adequate means for agents to convey affect.
REFERENCES


Derek J. Sollenberger and Munindar P. Singh. "upcoming jaamas article by author.". 2010.


APPENDIX
Appendix A

Developer Surveys

The surveys used in the Kokomo developer study can be found below.

Pre-Study Survey

1. On a scale of 1 (poor) to 5 (excellent) rate your experience / proficiency with the following tools: (1) Java, (2) Eclipse, (3) Android OS.

2. How many years of experience do you have developing with the following tools: (1) Java, (2) Eclipse, (3) Android OS.

3. Have you ever been employed as a software engineer? If so were you a part-time, intern, or full-time employee and how long were you employed?

4. What is the largest software application you have written individually? Please approximate the lines of code in the application and briefly describe the project.

5. What is the largest software application you have written as a member of a small team? Please approximate the lines of code in the application then briefly describe the project and the size of your team.

6. Have you ever taken a course (graduate, undergraduate, or both) specific to the following areas: (1) Artificial Intelligence, (2) Software Engineering.

7. Are you familiar with the concept of "Affective Computing"? If so please provide a brief
explanation of the term.

**In-Study Surveys**

1. Briefly describe what you accomplished during this work session?
2. How much time did you spend on the following tasks during this work session?
3. On a scale of 1 (extremely easy) to 9 (extremely difficult) rate the difficulty of the following areas during this work session.

**Post-Study Questionnaire**

1. Approximately how many hours did you spend working on the entire project?
2. On a scale of 1 (extremely easy) to 9 (extremely difficult) rate the difficulty of each item from Table A.1.
3. From easiest to hardest rank the difficulty of each item from Table A.1.
4. On a scale of 1 (no enjoyment) to 9 (extremely enjoyable) rate your level of enjoyment of each item from Table A.1.
5. From least to most enjoyable rank your favorite aspects of the project using each of the items from Table A.1.
6. What was the most enjoyable portion of the assignment? Explain.
7. What was the most frustrating portion of the assignment? Explain.
8. What improvements to the project would you recommend?
9. If you used Koko (Groups 2-3) answer the following questions.
   
   (a) On a scale of 1 (poor) to 5 (excellent) rate the following aspects of your experience with Koko: (1) Ease of Use, (2) Quality of Examples, (3) Quality of Documentation.
   
   (b) Describe the benefits and downsides of using Koko.
   
   (c) On a scale of 1 (useless) to 5 (extremely beneficial) rate how beneficial you found Koko to be in the development of your project.
Table A.1: Aspects of the Assignment

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gathering and utilizing the Exam Data</td>
</tr>
<tr>
<td>2</td>
<td>Gathering and utilizing the Environmental Data</td>
</tr>
<tr>
<td>3</td>
<td>Gathering and utilizing the Emotional Data</td>
</tr>
<tr>
<td>4</td>
<td>Composing the virtual TA heuristic</td>
</tr>
<tr>
<td>5</td>
<td>Writing and deploying the project using the Android Development Kit</td>
</tr>
</tbody>
</table>

**Post-Study Kokomo Questionnaire**

1. On a scale of 1 (hard) to 5 (easy) how difficult or challenging did you find each step of the methodology?

2. From easiest to hardest rank the difficulty of each methodology step when compared to the other steps.

3. Approximately how many hours did you spend on each step. For example, if a step only took 15 minutes put 0.25 and if it took 2 hrs and 30 minutes put 2.5.

4. Was any part of Step N confusing? Was there any aspect of this step that you felt could use more explanation? How would you change this step to make it more clear? (repeated for every step)