Auton Agent Multi-Agent Syst
DOI 10.1007/s10458-010-9160-3

Koko: an architecture for affect-aware games

Derek J. Sollenberger · Munindar P. Singh

Abstract The importance of affect in delivering engaging experiences in entertainment and educational games is well recognized. Yet, current techniques for building affect-aware games are limited, with the maintenance and use of affect in essence being handcrafted for each game. The Koko architecture describes a service-oriented middleware that reduces the burden of incorporating affect recognition into games, thereby enabling developers to concentrate on the functional and creative aspects of their applications. The Koko architecture makes three key contributions: (1) improving developer productivity by creating a reusable and extensible environment; (2) yielding an enhanced user experience by enabling independently developed games and other applications to collaborate and provide a more coherent user experience than currently possible; (3) enabling affective communication in multiplayer and social games. Further, Koko is intended to be used as an extension of existing game architectures. We recognize that complex games require additional third party libraries, such as game engines. To enable the required flexibility we define the interfaces of the Koko architecture in a formal manner, thereby enabling the implementation of those interfaces to readily adapt to the unique requirements of game's other architectural components and requirements.

Keywords Architecture · Affect · Games · Emotion

1 Introduction

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

This is a revised and extended version of an article that appears as [35].

D. J. Sollenberger (✉) · M. P. Singh
North Carolina State University, Raleigh, NC, USA
e-mail: djsollen@ncsu.edu

M. P. Singh
e-mail: singh@ncsu.edu

relevant event. Recognizing and reasoning about affect enables developers to create systems that demonstrate superior intelligence [14], enhanced user interfaces [2], and more effective learning environments [9,21].

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. For instance, the educational game Prime Climb [5] employs a virtual character that offers advice in order to guide the player through the game. The manner in which the virtual character interacts with the player is based not only on the player's current progress but also on the player's estimated affective state. The inclusion of affect enables the virtual character to better understand the player in order to help guide them through the game while meeting the specified learning objectives.

Although many current affect-aware games and applications are leading the way in their respective research areas, they all suffer from a few limitations. First, existing affect-aware 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 approach presupposes the game developer be an expert in the domain of affect and has resulted in games and applications that are tied to a specific affect model.

Further, there is no interoperability or continuity between affect-aware games. For instance, most modern affect-aware games and their underlying affect model implementations are based on the discipline of appraisal theory [34]. 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. A current and common practice has been to copy and edit a model from an existing game (and, occasionally, to build from scratch) to meet the specifications of a new domain.

Such approaches are clearly not best software engineering practices. 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. To this end, we introduce Koko, a multiagent and service-oriented architecture for middleware, which enables the prediction of a player's affective state. Koko is not another model of emotions but a (logically described) middleware 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 modules 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.

It is worth remarking here that Koko is intended to be used to support affective models and applications that seek to recognize emotion in a human user. Although it is possible to use Koko to model the emotions of virtual nonplaying characters, many of Koko's benefits most naturally apply when human users are involved.

This article demonstrates two key aspects of Koko, namely its architectural improvements and its ability to be coupled with existing game application architectures.

1.1 Architectural improvements

One of our primary motivations is to identify common components in affect-aware games and then absorb those components into Koko. The two primary components of affect-aware games are the *affect model* and *physical sensors*. The affect model is the part of the affect-aware 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 affect-aware applications use physical sensors to provide additional information about both the user and the user's environment [29]. 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). In this connection, it is worth mentioning the emergence of industry standards groups such as the Continua Health Alliance [6], which standardizes a variety of personal sensors of direct value to medical and healthful living applications.

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 Sect. 3.

1.1.1 Benefits

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. In Koko, the game logic and the affect model are treated 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 type 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 is thereby unaware of other games employed by that user. By contrast, Koko-based games can share common affective data through Koko. This can reduce each application's overhead of initializing the user's affective state for each session, as well as provide 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 improves the value of each game. As a use case, consider a student playing 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 (or the user's agent) 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.

1.1.2 Significance

Any software architecture is motivated by improvements in features such as modularity and maintainability: you can typically achieve the same functionality through more complex or less elegant means [32]. Of course, an improved architecture facilitates accessing new functionality: in our case, the sharing of affective information and the design of social applications.

Additionally, architectural improvements can have scientific significance. For example, in the history of artificial intelligence, separating knowledge bases from problem solving has not only improved developer productivity, but has also led to greater clarity and improvements in the concepts of knowledge representation and problem solving. Koko takes similar steps with respect to affect, especially by supporting cross-application and cross-user affective interactions.

1.2 Architectural connectors

The Koko architecture as a stand-alone unit is not sufficient to build most complex games. 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 a 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 our 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 affect-aware 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. We formally define both the configuration and runtime interfaces in the mathematical Z notation [37], which is an established way of describing

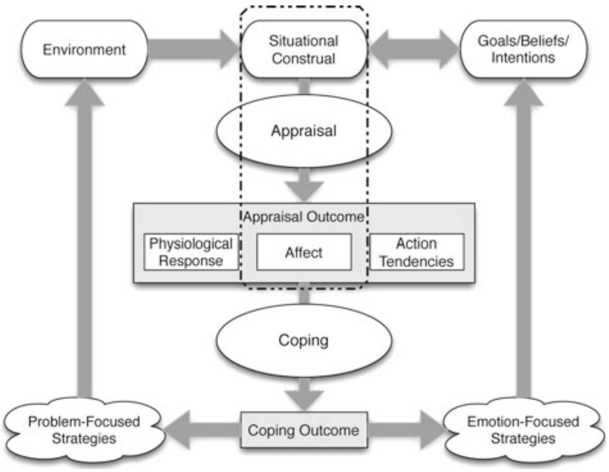


Fig. 1 Components and process of appraisal theory [34]

software components and architectures and has been successfully applied in agent software as well [1].

We have carefully ensured that the Koko architecture is *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 Sect. 6.3, we elaborate on three means of deploying Koko and discuss the relative merits of each.

1.3 Paper organization

The remainder of this article is arranged as follows. Section 2 reviews appraisal theory. Section 3 describes the progression in programming models for affective games. Section 4 provides a detailed description of Koko's components. Section 5 describes the formal interfaces between Koko and other architectures. Section 6 demonstrates the claimed merits of the Koko architecture. Finally, Sect. 7 discusses other topics of interest and future work.

2 Background

Smith and Lazarus [34] developed the cognitive-motivational-emotive model, which has become the baseline for current appraisal theory models. As Fig. 1 shows, this model 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 [18].

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 [22]. 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 Sect. 5.1.1.

In appraisal theory, a situational construal combines the environment (facts about the world) and the internal state of the individual (goals and beliefs) and produces the individual's perception of the world, which then drives the appraisal and provides an appraisal outcome. This appraisal outcome is made up of multiple facets, of which the central facet is *Affect*, that correspond to the individual's current emotions. For practical purposes, *Affect* can be interpreted as a set of discrete states, each with an associated intensity. For instance, the result of an appraisal could be that you are simultaneously *happy* with an intensity of α and *proud* with an intensity of β. 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 Sect. 4.1.

Next, we look at three existing appraisal theory approaches for modeling emotion. The first is a classic approach that 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 nonplaying characters whereas CARE concentrates on measuring human emotional states.

OCC

Ortony et al. [27] 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 OCCs set of emotions have turned out to cover a broad portion of the affective space. Elliott [8] 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 [25] to extend Smith and Lazarus' model for applications involving nonplaying characters [15]. 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 [16]. Further, EMA was one of the first domain independent models of affect [17] 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 [24]. 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 article. Other probabilistic models similar to CARE, such as the one used in the Prime Climb [5] game would also be suitable reference models for Koko. We have chosen CARE as our reference because it has an easily understandable description and its implementation was readily accessible.

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 the proposed improvements.

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 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 [31] 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

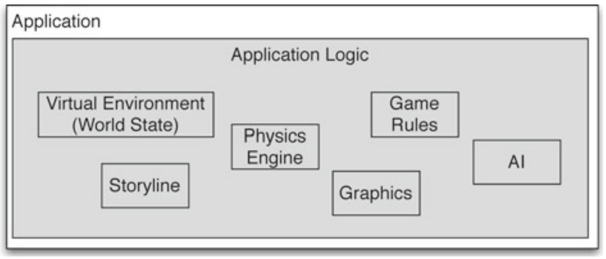


Fig. 2 Monolithic programming model

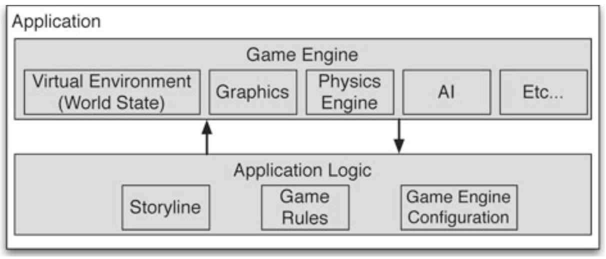


Fig. 3 Component-based programming model

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

The component-based programming model addresses many of the challenges posed by the monolithic model at the cost of total control [31]. As Fig. 3 shows, in a component-based model, the developer is responsible only for the application logic whereas 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 avoids having 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 developers to focus on the creative aspects of their game. A developer then need only configure the game engine to customize game-specific features. Doing so 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 vs. 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

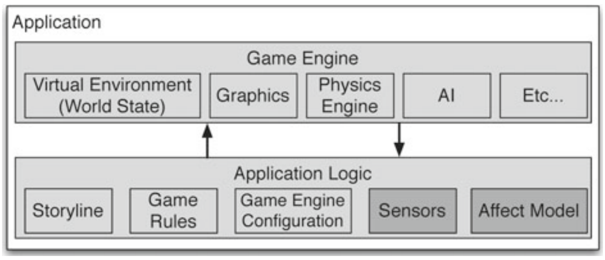


Fig. 4 Affective programming model

the leading game engines are the Unreal Engine [10] and the Source Engine [39], both of which are designed for use on personal computers and video game consoles and both are architected using the component-based model.

Affect-aware

As we discussed in Sect. 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 (Fig. 4). Examples of this affect-aware programming model in action are the games Treasure Hunt [24], Crystal Island [19], and FearNot [7].

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 a growing number of 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 an educational game, a heart rate sensor may be relevant only to the affect model, but in a physical fitness game it could be appropriate to also display the heart rate information directly to the user.

Koko

The Koko programming model makes improvements to the affect-aware programming model, much in the same manner as the component-based model improves over the monolithic model. Existing affect-aware applications follow a monolithic style 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 [32].

As we discussed in Sect. 1.1 the introduction of Koko provides the benefits of an increase in developer productivity, an enhanced user experience, and the enabling of affective social applications. Koko abstracts out the common components in affect-aware games, namely, the

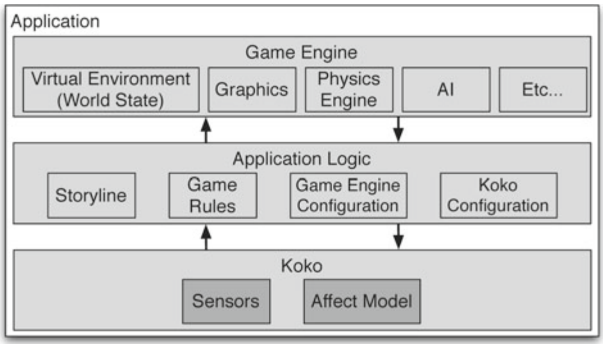


Fig. 5 Koko programming model

affect model and physical sensors (Fig. 5). Using the Koko programming model, developers need not interface with those components directly, which is often a tedious and nontrivial task. Instead, they can leverage the programming efforts of other programmers employing Koko. For instance, the programming effort required to gather data from a sensor can be performed by one developer and registered in Koko, which makes the sensor eligible for reuse by other Koko-based applications.

4 Components of Koko

Figure 6 shows Koko's general architecture using arrows to represent data flow. The following subsections provide details on each of Koko's major components. After we explain the groundwork of the architecture then we elaborate on the formal interface definitions in the following section.

Koko hosts an active computational entity or *agent* for each user. In particular, Koko requires and ensures that there is exactly one agent per user, 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 Fig. 6 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.

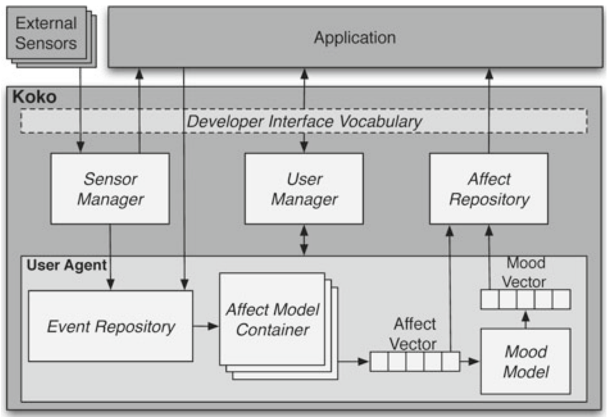


Fig. 6 Koko architectural overview

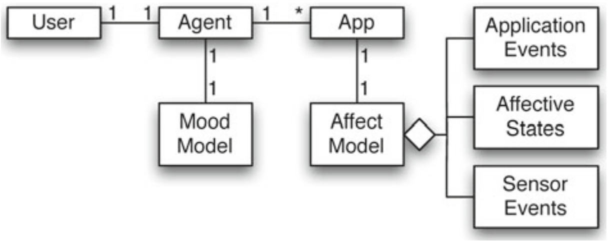


Fig. 7 Main Koko entities

4.1 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 Fig. 7 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 instance 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) and 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 instance by populating predictive data structures with affective knowledge. These affect model instances are built, executed, and maintained by the container using the Weka toolkit [41]. 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).

4.2 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 [28] 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 [7] on the FearNot! model of emotions. 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.

4.3 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 the applications and to the affect and mood models of the Koko user 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 read.

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 an 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.

4.4 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 vocabulary.

4.5 Sensor manager

Information about a user's physical state (for example, heart rate and skin conductivity) 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 introducing new sensor types 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.

4.6 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 [30], but the details of such policies lie outside the scope of this paper.

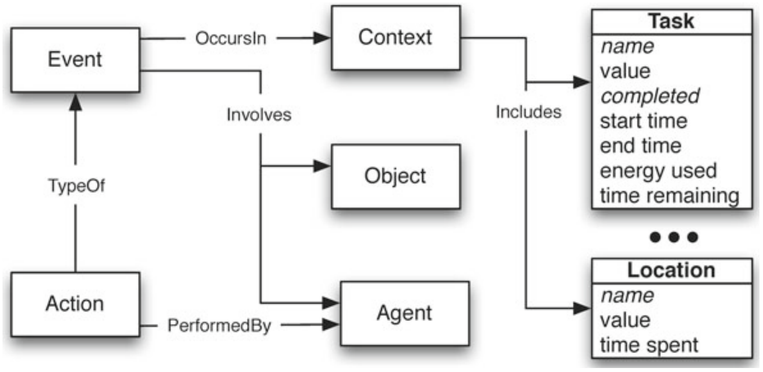


Fig. 8 Event ontology example

4.7 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 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 Elliott [8]. 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 specifies the type of data that applications and sensors will be sending. The event definition is constructed by selecting terms from the ontology that are relevant 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.

Koko comes preloaded with an event ontology (partially shown in Fig. 8) 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 on 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).

5 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 introduce our notation. The interfaces are specified using the mathematical notation language Z [37] 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 Sect. 4. 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, "Δ" denotes an operation updates the schema, while "Ξ" denotes an operation that does 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.

$$[AGENT\_ID, APPLICATION\_ID, AFFECT\_MODEL,\\ AFFECT\_INTENSITY, AFFECTIVE\_STATE, EVENT]$$

$$VECTOR\_TYPE ::= application \mid mood$$

KokoAgent

$$\begin{array}{l}
agentID : AGENT\_ID \\
moodVector : AffectVector \\
applications : \mathbb{P}\, KokoApplication \\
getApplication : APPLICATION\_ID \rightarrowtail KokoApplication \\
\hline
applications \neq \emptyset \\
moodVector.type = mood
\end{array}$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{KokoApplication}}\\
\hline
applicationID : APPLICATION\_ID\\
affectModel : AFFECT\_MODEL\\
affectVectors : \mathbb{P}\,AffectVector\\
events : \mathbb{P}\,EVENT\\
predictOutcome : EVENT \rightarrowtail\!\!\!\!\!+ \; AffectVector\\
\hline
affectModel \neq \emptyset\\
\forall\, v : affectVectors \bullet v.type = application\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AffectVector}}\\
\hline
time : \mathbb{Z}\\
type : VECTOR\_TYPE\\
affectiveStates : \mathbb{P}\,AFFECTIVE\_STATE\\
intensities : \mathbb{P}\,AFFECT\_INTENSITY\\
getIntensity : AFFECTIVE\_STATE \rightarrowtail\!\!\!\!\!+ \; AFFECT\_INTENSITY\\
\hline
\end{array}
$$

5.1 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 to declare 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.

5.1.1 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 Sect. 5.1.2.

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.

$$
\begin{array}{l}
\textit{AddApplicationEvent} \\
\hline
\Delta \textit{KokoAgent} \\
\textit{userID}? : \textit{AGENT\_ID} \\
\textit{applicationID}? : \textit{APPLICATION\_ID} \\
\textit{applicationEvent}? : \textit{EVENT} \\
\hline
\textit{userID}? = \textit{agentID} \\
(\textbf{let}\ \textit{application} == \textit{getApplication}(\textit{applicationID}?) \bullet \\
\qquad\qquad\qquad \textit{getApplication}(\textit{applicationID}?) \neq \emptyset) \\
\textit{application.events}' = \textit{application.events} \cup \textit{applicationEvent}? \\
\hline
\end{array}
$$

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 instance. 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.

$$
\begin{array}{l}
\textit{GetCurrentAffectVector} \\
\hline
\Xi \textit{KokoAgent} \\
\textit{userID}? : \textit{AGENT\_ID} \\
\textit{applicationID}? : \textit{APPLICATION\_ID} \\
\textit{affectResult}! : \textit{AffectVector} \\
\hline
\textit{userID}? = \textit{agentID} \\
(\textbf{let}\ \textit{application} == \textit{getApplication}(\textit{applicationID}?) \bullet \\
\qquad\qquad\qquad \textit{getApplication}(\textit{applicationID}?) \neq \emptyset) \\
(\textbf{let}\ \textit{maxTime} == \max\{\forall\, v.\textit{time} : \textit{application.affectVectors}\}) \\
\textit{affectResult}! = \{v : \textit{application.affectVectors} \mid v.\textit{time} = \textit{maxTime}\} \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{GetAffectVector}} \\
\Xi \mathit{KokoAgent} \\
\mathit{userID?} : \mathit{AGENT\_ID} \\
\mathit{applicationID?} : \mathit{APPLICATION\_ID} \\
\mathit{startTime?} : \mathbb{Z} \\
\mathit{endTime?} : \mathbb{Z} \\
\mathit{affectResults!} : \mathbb{P}\, \mathit{AffectVector} \\
\hline
\mathit{userID?} = \mathit{agentID} \\
(\mathbf{let}\ \mathit{application} == \mathit{getApplication}(\mathit{applicationID?}) \bullet \\
\qquad\qquad \mathit{getApplication}(\mathit{applicationID?}) \neq \emptyset) \\
\mathit{affectResults!} = \{ v : \mathit{application.affectVectors} \mid v.\mathit{time} \geq \mathit{startTime?} \land \\
\qquad\qquad\qquad\qquad v.\mathit{time} \leq \mathit{endTime?} \} \\
\hline
\end{array}$$

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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{GetPredictedAffectVector}} \\
\Xi \mathit{KokoAgent} \\
\mathit{userID?} : \mathit{AGENT\_ID} \\
\mathit{applicationID?} : \mathit{APPLICATION\_ID} \\
\mathit{predictedEvent?} : \mathit{EVENT} \\
\mathit{affectResult!} : \mathit{AffectVector} \\
\hline
\mathit{userID?} = \mathit{agentID} \\
(\mathbf{let}\ \mathit{application} == \mathit{getApplication}(\mathit{applicationID?}) \bullet \\
\qquad\qquad \mathit{getApplication}(\mathit{applicationID?}) \neq \emptyset) \\
\mathit{affectResult!} = \mathit{application.predictOutcome}(\mathit{predictedEvent}) \\
\hline
\end{array}$$

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, becomes frustrated playing it, and 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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{GetMood}} \\
\Xi \mathit{KokoAgent} \\
\mathit{userID?} : \mathit{AGENT\_ID} \\
\mathit{moodResult!} : \mathit{AffectVector} \\
\hline
\mathit{userID?} = \mathit{agentID} \\
\mathit{moodResult!} = \mathit{moodVector} \\
\hline
\end{array}$$

5.1.2 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 identify key pieces of information and supply that information to Koko using the following interface:

$$[SENSOR\_ID, MODEL\_ID]$$

$$\begin{array}{|l}
\textit{InitApplication} \\
\Delta KokoAgent \\
affectiveStates? : \mathbb{P}\, AFFECTIVE\_STATE \\
eventDefinitions? : \mathbb{P}\, EVENT \\
sensorIDs? : \mathbb{P}\, SENSOR\_ID \\
modelID? : MODEL\_ID \\
applicationID! : APPLICATION\_ID \\
newApplication : \{\mathbb{P}\, AFFECTIVE\_STATE\} \times \{\mathbb{P}\, EVENT\} \times \\
\qquad\qquad \{\mathbb{P}\, SENSOR\_ID\} \times MODEL\_ID \rightarrowtail KokoApplication \\
\hline
affectiveStates? \neq \emptyset \\
eventDefinitions? \neq \emptyset \\
modelID? \neq \emptyset \\
(\textbf{let}\ application == newApplication(affectiveStates?, eventDefinitions?, \\
\qquad\qquad\qquad\qquad sensorIDs?, modelID?)) \\
applications' = applications \cup application \\
applicationID! = application.applicationID
\end{array}$$

Additional Z primitives are needed to describe the configuration, 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 may effect the user's emotional status, whereas the time until the application needs to garbage collect its data structures would be 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.

$$\begin{array}{|l}
\textit{ModelTraining} \\
\Delta \textit{KokoAgent} \\
\textit{userID}? : \textit{AGENT\_ID} \\
\textit{applicationID}? : \textit{APPLICATION\_ID} \\
\textit{applicationEvent}? : \textit{EVENT} \\
\textit{classifier}? : \textit{AffectVector} \\
\hline
\textit{userID}? == \textit{agentID} \\
(\textbf{let}\ \textit{application} == \textit{getApplication}(\textit{applicationID}?) \bullet \\
\qquad\qquad\qquad \textit{getApplication}(\textit{applicationID}?) \neq \emptyset) \\
\textit{application.affectModel}' = \textit{application.affectModel} \cup \\
\langle \textit{applicationEvent}?, \textit{classifier}? \rangle
\end{array}$$

5.2 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 an 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 an instance of Koko, which adds it to the appropriate resource pool. The agents operating within that Koko instance 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 is implemented. To enable a wide variety of models Koko places only one precondition on their construction, which 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, a 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 affect model plugin. The schema introduces one new primitive, called MODEL_LOGIC. The MODEL_LOGIC is a primitive (as defined by Z) that 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. An affect model designer uses the developer interface vocabulary to specify the inputs that the model can accept. Specifically, the emotion ontology is used for describing the *supportedAffectiveStates* and the event ontology for describing the *supportedEventDefinitions*. The *modelID* is used to encode not only the name of the model but also its description, version number, and other auxiliary information.

$[\textit{MODEL\_LOGIC}]$

$$
\begin{array}{|l}
\hline
\textit{AffectModelDescription} \\
\textit{modelID} : \textit{MODEL\_ID} \\
\textit{supportedAffectiveStates} : \mathbb{P}\,\textit{AFFECTIVE\_STATE} \\
\textit{supportedEventDefinitions} : \mathbb{P}\,\textit{EVENT} \\
\textit{logic} : \textit{MODEL\_LOGIC} \\
\hline
\textit{modelID} \neq \emptyset \\
\textit{logic} \neq \emptyset \\
\hline
\end{array}
$$

Applications choose which affect model to used based on the available *AffectModelDescriptions*. Then agents whose users employ that application use the description to generate an *AffectModelInstance*. The resulting *AffectModelInstance* replaces the AFFECT_MODEL primitive defined earlier in the section. The difference between the *AffectModelDescription* and *AffectModelInstance* is the same as the difference in programming between an object class and an instance of that class. The transition from *AffectModelDescription* to *AffectModelInstance* is easy to describe verbally, the notation in 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 MODEL_LOGIC using the application's *affectiveStates* and *eventDefinitions* (like parameters of an object's constructor), which then produces the *performAppraisal* function.

$$
\begin{array}{|l}
\hline
\textit{AffectModelInstance} \\
\textit{description} : \textit{AffectModelDescription} \\
\textit{affectiveStates}? : \mathbb{P}\,\textit{AFFECTIVE\_STATE} \\
\textit{eventDefinitions}? : \mathbb{P}\,\textit{EVENT} \\
\textit{events} : \mathbb{P}\,\textit{EVENT} \\
\textit{appraisalOutcomes} : \mathbb{P}\,\textit{AffectVector} \\
\textit{performAppraisal} : \textit{EVENT} \rightarrowtail \textit{AffectVector} \\
\hline
\forall e : \textit{eventDefinitions} \bullet e \in \textit{description.supportedEventDefinitions} \\
\forall v : \textit{affectiveStates} \bullet v.\textit{affectiveStates} \in \textit{description.supportedAffectiveStates} \\
\forall e : \textit{events} \bullet e \in \textit{eventDefinitions} \\
\forall v : \textit{appraisalOutcomes} \bullet v.\textit{affectiveStates} \in \textit{affectiveStates} \\
\hline
\end{array}
$$

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 that compels the agent to conduct an appraisal and produce an affective response. The formal notation for such an interaction is as follows.

$$
\begin{array}{|l}
\hline
\textit{Appraisal} \\
\Delta \textit{AffectModelInstance} \\
\textit{event}? : \textit{EVENT} \\
\textit{appraisalResult}! : \textit{AffectVector} \\
\hline
\textit{event}? \neq \emptyset \\
\textit{appraisalResult}! = \textit{performAppraisal}(\textit{event}?) \\
\textit{events}' = \textit{events} \cup \textit{event}? \\
\textit{appraisalOutcomes}' = \textit{appraisalOutcomes} \cup \textit{appraisalResult}! \\
\hline
\end{array}
$$

The affect model can also support additional nonmandatory interfaces. For instance, it can choose to support affect prediction in order to enable the *GetPredictedAffectVector* interface or if the model supports learning it can implement the *ModelTraining* interface. It should also be noted that an affect model may access all its parent agent's resources and inter-agent messaging capabilities as described in Sect. 4.

6 Evaluation

Our evaluation mirrors our claimed contributions, namely, the benefits of the Koko architecture. First, we evaluate the contributions of the architecture in itself. Subsequently, we demonstrate the usefulness of the architecture with case studies of both single and multiplayer games. Finally, we elaborate on the relative merits of three different means of deploying Koko.

6.1 Architectural benefits

A software architecture is primarily motivated not by functionality but by so-called "ilities" or nonfunctional properties [11]. 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 affect model installed in Koko as a plugin.

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 intraagent and *cross-user* or inter-agent communication. For a social or multiplayer 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.

6.2 Case studies

In this section, we present two case studies that demonstrate Koko in operation. The first study is the creation of a new social game, called booST, which illustrates the social and multiplayer aspects of Koko. The second study is the re-creation of the affect-enabled Treasure Hunt game, which helps us illustrate the differences between the Koko and CARE architectures.

6.2.1 Koko implementation

Both booST and Treasure Hunt employ the same instance of the Koko middleware. Therefore, Koko maintains only one agent for per user even if the user participates in both 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, both booST and Treasure Hunt 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 Sect. 5.2. 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 *Affect-ModelDescription*. 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 1 min when it is performing an appraisal.

6.2.2 booST

The subject of our first case study is a social, physical health application with affective capabilities, called booST. In its present implementation, to operate, booST requires a mobile phone running Google's Android mobile operating system that is equipped with a GPS sensor. Notice that while booST takes advantage of the fact that the sensor and application run on the same device, this is not a restriction imposed by Koko.

The purpose of booST is to promote positive physical behavior in young adults by enhancing their social network with affective capabilities and interactive activities. As such, booST utilizes the OpenSocial platform [26] 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 its use of the *energy levels* and *emotional status* of its users.

Each user is assigned an *energy level* that is computed using simple heuristics from data retrieved from the GPS. One can think of energy in this sense as examples of an ad hoc, game-specific function of the user's emotional state. Additionally, each user is assigned an *emotional status* generated from the affect vectors retrieved from Koko. The emotional status is represented as a number ranging from 1 to 10. The higher the number the happier the individual is reported to be. A user's energy level and emotional status are made available both to the user and to members of the user's social circle.

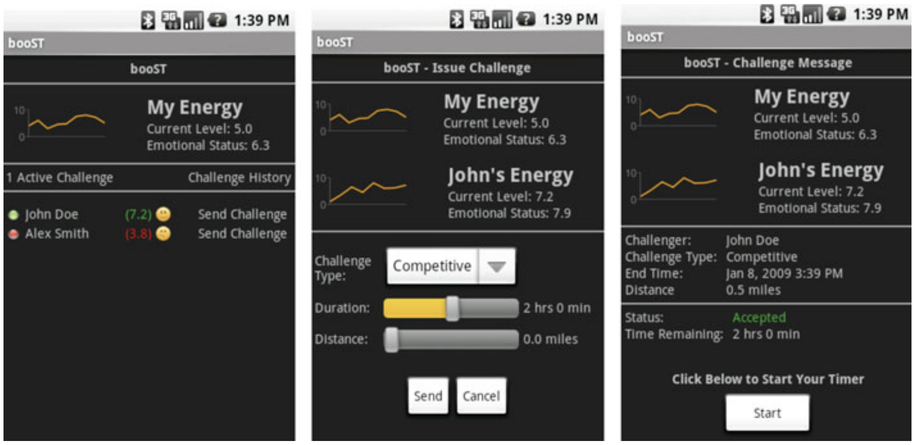
To promote positive physical behavior, booST supports interactive physical activities among members of a user's social circle. The activities are classified as either competitive or cooperative. Each type of activity employs the GPS to determine the user's progress toward achieving its goal. The difference 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 both to win (equally).

Koko's primary function in booST is to maintain the affect model that booST employs to generate the user's emotional status. In general, the application provides the data about its environment. In the case of booST, the user's environment consists of the user's social interactions and the user's participation in the booST activities. Koko passes this environmental information to the appropriate user agent, who processes the data and returns the appropriate emotional status. Further, Koko enables the exchange of affective state with the members of a user's social circle. This interaction can be seen in Fig. 9 in the emotions 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, and make the user *sad* (in empathy, if you like).

6.2.3 Treasure hunt

We showed above how a new application, such as booST, can be built from scratch 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 that demonstrates the original CARE model [24] wherein the user controls a character to carry out some pedagogical tasks in a virtual


1:39 PM
booST
booST
My Energy
Current Level: 5.0
Emotional Status: 6.3
1 Active Challenge   Challenge History
John Doe (7.2) Send Challenge
Alex Smith (3.8) Send Challenge

1:39 PM
booST
booST - Issue Challenge
My Energy
Current Level: 5.0
Emotional Status: 6.3
John's Energy
Current Level: 7.2
Emotional Status: 7.9
Challenge Type: Competitive
Duration: 2 hrs 0 min
Distance: 0.0 miles
Send Cancel

1:39 PM
booST
booST - Challenge Message
My Energy
Current Level: 5.0
Emotional Status: 6.3
John's Energy
Current Level: 7.2
Emotional Status: 7.9
Challenger: John Doe
Challenge Type: Competitive
End Time: Jan 8, 2009 3:39 PM
Distance 0.5 miles
Status: Accepted
Time Remaining: 2 hrs 0 min
Click Below to Start Your Timer
Start

Fig. 9 booST buddy list and activities screenshots

Table 1 Three phases of CARE affect models

No.	Description
1	Gather environmental data and emotional classifiers
2	Perform supervised ML techniques on the data and classifiers
3	Given environmental data produce emotional probabilities

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 user's environment 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 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.

Koko improves on the original CARE architecture by (1) eliminating the need for the application to keep a record of its environmental data and (2) performing the learning online. As a result, the Koko-based TH can move fluidly from phase 1 to phase 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 though possible in principle, results in an application restart to inject the new affect model. Thus it is quite constrained.

The foregoing improvements to TH can be viewed by analogy to how an iterative software engineering approach compares with 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 phase of the process at the time you enter that phase. If the data in a previous phase changes, you can return to a previous phase but at a high cost. Koko follows a more iterative approach by removing the assumption that all the information will be available ahead of time and by ensuring that transitions to a previous phase incur no additional cost. As a result, Koko yields a more efficient application architecture of the TH game than the original CARE architecture does.

6.3 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 the above two case studies, 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 (Fig. 10). 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 yields 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 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 (Fig. 11). 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 disadvantage that they cannot make guarantees about latency. Latency is uncertain because

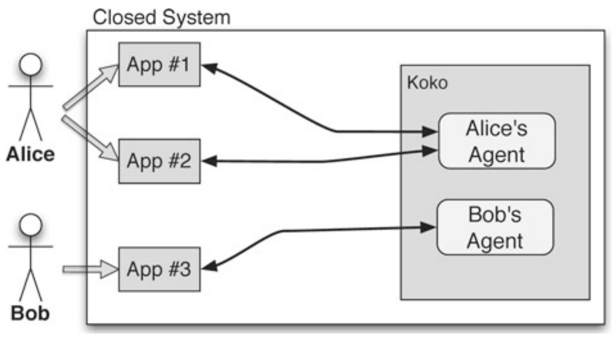


Fig. 10 Closed system deployment

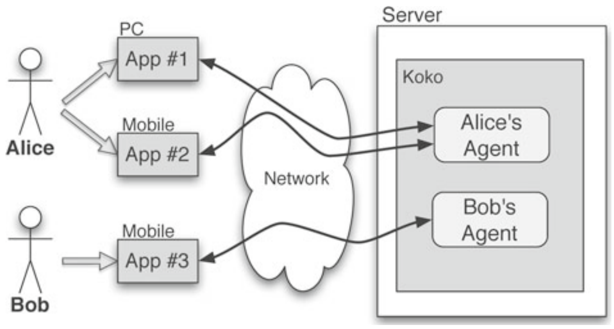


Fig. 11 Global system deployment

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 Fig. 12 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

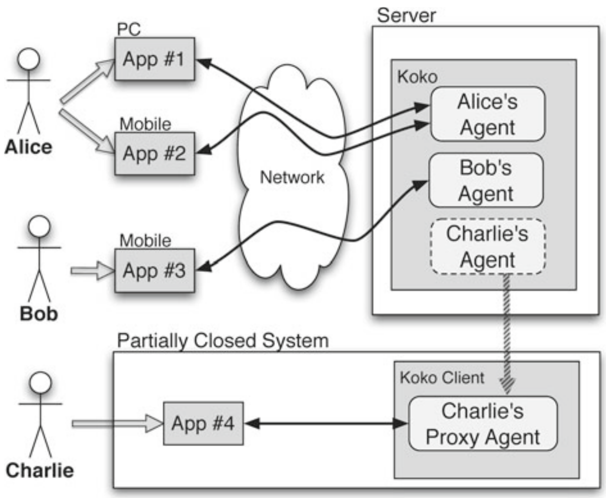


Fig. 12 Hybrid system deployment with performance proxy

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.

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.

7 Discussion

This paper shows how software architecture principles can be used to specify a middleware for social affective computing that cooperates with existing game engines. If affective computing is to have the practical impact that many hope it will, advances in software architecture are crucial. Further, the vocabulary of events and context attributes introduced here can form the basis of a standard approach for building and hosting affective applications.

Game engine integration

Due to the social, multiplayer, and multiapplication 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 and unmarshalling objects from the engine's data structures to those supported by Koko.

Affect modeling

Existing appraisal theory applications are developed in a monolithic manner [8] 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 [28], 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) [23]. For example, Conati [4] developed a CB-AUM style model using dynamic Bayesian networks as a means to model student emotions in pedagogical games. 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. [12]. 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 [20] and NonKin Village [33] applications could access the affect state of the player and use that information to enhance the agent's decision making process.

Affective gaming

Commercial games, such as the popular series Mass Effect [3] and Assassin's Creed [38], 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 [24] and Prime Climb [5] primarily because we lack visibility into the modeling techniques employed in commercial games.

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 [40]. 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.

Enhanced social networking

Human interactions rely upon social intelligence [13]. 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.

Future work

Koko opens up promising areas for future research. In particular, we would like to further study the challenges of sharing affective data between applications and users. In particular we are interested in exploring the types of communication that can occur between affective agents.

Additionally, we have developed a methodology for building affect-aware, social applications [36]. We are interested in refining and enhancing that methodology for gaming applications that incorporate affect.

Acknowledgments Munindar Singh's effort was partially supported by National Science Foundation Grant #0910868.

References

1. Ashri, R., Luck, M., & d'Inverno, M. (2002). Infrastructure support for agent-based development. In *Foundations and applications of multi-agent Systems*, volume 2403 of *LNCS* (pp. 1542–1558). Berlin: Springer.
2. Bickmore, T., Mauer, D., Crespo, F., & Brown, T. (2007). Persuasion, task interruption and health regimen adherence. In *Second International Conference on Persuasive Technology for Human Well-Being (Persuasive)*, April 2007, Stanford, CA.
3. Bioware Group a division of Electronic Arts Inc. Mass Effect (2010). http://bioware.com.
4. Conat, C., & Maclaren, H. (2005). Data-driven refinement of a probabilistic model of user affect. In *User modeling 2005*, volume 3538 of *LNCS* (pp. 40–49), August 2005. Berlin: Springer.
5. Conati, C., & Maclaren, H. (2009) Modeling user affect from causes and effects. In *Proceedings of the 17th International Conference on User Modeling, Adaptation, and Personalization*, volume 5535 of *LNCS* (pp. 4–15), September 2009. Berlin: Springer.
6. Continua (2009). Continua health alliance. http://www.continuaalliance.org.
7. Dias, J., & Paiva, A. (2005). Feeling and reasoning: A computational model for emotional characters. In *Progress in artificial intelligence*, volume 3808 of *LNCS* (pp. 127–140). Berlin: Springer.
8. Elliott, C. (1992). *The affective reasoner: A process model of emotions in a multi-agent system*. PhD Thesis, Northwestern University, Evanston, IL.
9. Elliott, C., Rickel, J., & Lester, J. (1999). Lifelike pedagogical agents and affective computing: An exploratory synthesis. *Artificial Intelligence Today, LNAI, 1600*, 195–212.
10. Epic Games Inc. (2009). Unreal engine. http://www.epicgames.com.
11. Filman, R. E., Barrett, S., Lee, D. D., & Linden, T. (2002). Inserting ilities by controlling communications. *Communications of the ACM, 45*(1), 116–122.
12. Gilroy, S., Cavazza, M., Niiranen, M., André, E., Vogt, T., Urbain, J., Benayoun, M., Seichter, H., & Billinghurst, M. (2009). PAD-based multimodal affective fusion. *Proceedings of the Third International Conference on Affective Computing and Intelligent Interaction* (pp. 1–8). Amsterdam: IEEE.
13. Goleman, D. (2006). *Social intelligence: The new science of human relationships*. New York, NY: Bantam Books.
14. Gratch, J., & Marsella, S. (2003). Fight the way you train: The role and limits of emotions in training for combat. *The Brown Journal of World Affairs, X*(1), 63–76.
15. Gratch, J., & Marsella, S. (2004). A domain-independent framework for modeling emotion. *Journal of Cognitive Systems Research, 5*(4), 269–306.
16. Gratch, J., Mao, W., & Marsella, S. (2006). *Modeling social emotions and social attributions*. Cambridge, UK: Cambridge University Press.
17. Gratch, J., Marsella, S., Wang, N., & Stankovic, B. (2009). Assessing the validity of appraisal-based models of emotion. *Proceedings of Third International Conference on Affective Computing and Intelligent Interaction* (pp. 1–8). Amsterdam: IEEE.
18. Lazarus, R. S. (1991). *Emotion and adaptation*. New York, NY: Oxford University Press.
19. Lee S., McQuiggan S.W., & Lester, J. C. (2007). *Proceedings of the Eleventh International Conference on User Modeling*, volume 4511 of *LNCS*, inducing user affect recognition models for task-oriented environments (pp. 380–384). Berlin: Springer.

20. Lim, M. Y., Dias, J., Aylett, R., & Paiva, A. (2009). Intelligent npcs for education role play game. In F. Dignum, B. Silverman, J. Bradshaw, & W. van Doesburg (Eds.), *Proceedings of the First International Workshop on Agents for Games and Simulations*, volume 5920 of *LNAI* (pp. 107–118). Berlin: Springer.
21. Marsella, S., Johnson, W. L., & LaBore, C. (2000). Interactive pedagogical drama. In *Proceeding of Fourth International Conference on Autonomous Agents* (pp. 301–308). Montreal, QC, Canada.
22. Marsella, S., Gratch, J., Wang, N., & Stankovic, B. (2009). Assessing the validity of a computational model of emotional coping. *Proceedings of the Third International Conference on Affective Computing and Intelligent Interaction* (pp. 1–8). Amsterdam: IEEE.
23. Martinho, C., Machado, I., & Paiva, A. (2000). A cognitive approach to affective user modeling. In *Affective interactions*, volume 1814 of *LNAI* (pp. 64–75). Berlin: Springer.
24. McQuiggan, S., Lee, S., & Lester, J. (2006). Predicting user physiological response for interactive environments: An inductive approach. In J. E. Laird & J. Schaeffer (Eds.), *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference* (pp. 60–65). Stanford, CA: AAAI Press.
25. Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.
26. OpenSocial Foundation (2009). Opensocial. http://www.opensocial.org.
27. Ortony, A., Clore, G. L., & Collins, A. (1988). *The cognitive structure of emotions*. Cambridge, MA: Cambridge University Press.
28. Picard, R. W. (1997). *Affective computing*. Cambridge, MA: MIT Press.
29. Picard, R. W., & Healey, J. (1997). Affective wearables. *Personal and Ubiquitous Computing, 1*(4), 231–240.
30. Reynolds, C., & Picard, R.W. (2004). Affective sensors, privacy, and ethical contracts. In *Proceedings of the 22nd International Conference on Human Factors in Computing Systems* (pp. 1103–1106). New York: ACM.
31. Rollings, A., & Morris, D. (2000). *Game architecture and design*. Scottsdale, AZ: Coriolis.
32. Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Upper Saddle River, NJ: Prentice-Hall.
33. Silverman, B., Chandrasekaran, D., Weyer, N., Pietrocola, D., Might, R., & Weaver, R. (2009). Nonkin village: A training game for learning cultural terrain and sustainable counter-insurgent operations. In F. Dignum, B. Silverman, J. Bradshaw, & W. van Doesburg (Eds.), *Proceedings of the First International Workshop on Agents for Games and Simulations*, volume 5920 of *LNAI* (pp. 135–154). Berlin: Springer.
34. Smith, C., & Lazarus, R. (1990). Emotion and adaptation. In L. A. Pervin & O. P. John (Eds.), *Handbook of personality: Theory and research* (pp. 609–637). New York, NY: Guilford Press.
35. Sollenberger, D. J., & Singh, M. P. (2009). Architecture for affective social games. In F. Dignum, B. Silverman, J. Bradshaw, & W. van Doesburg (Eds.), *Proceedings of the First International Workshop on Agents for Games and Simulations*, volume 5920 of *LNAI* (pp. 135–154). Berlin: Springer.
36. Sollenberger, D. J., & Singh, M. P. (2011). Methodology for engineering affective social applications. In M. P. Gleizes, & J. J. Gomez-Sanz (Eds.), *Proceedings of the Tenth International Workshop on Agent-Oriented Software Engineering*, volume 6038 of *LNCS* (pp. 97–109). Berlin: Springer.
37. Spivey, J. M. (1992). *The Z notation: A reference manual*. London, UK: Prentice Hall.
38. Ubisoft Entertainment Inc. (2010). Assassin's Creed. http://www.ubi.com.
39. Valve Software Inc. (2009). Source engine. http://source.valvesoftware.com.
40. Wagner J., André E., Jung F. (2009). Smart sensor integration: A framework for multimodal emotion recognition in real-time. *Proceedings of the Third International Conference on Affective Computing and Intelligent Interaction*. Amsterdam: IEEE.
41. Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* (2nd ed.). San Francisco, CA: Morgan Kaufmann.