Australas Phys Eng Sci Med (2017) 40:739–749
DOI 10.1007/s13246-017-0559-x

TECHNICAL PAPER

A software framework for real-time multi-modal detection of microsleeps

Simon J. Knopp[1,2] · Philip J. Bones[1] · Stephen J. Weddell[1] · Richard D. Jones[1,2]

Received: 19 May 2016 / Accepted: 14 May 2017 / Published online: 1 June 2017

Abstract A software framework is described which was designed to process EEG, video of one eye, and head movement in real time, towards achieving early detection of microsleeps for prevention of fatal accidents, particularly in transport sectors. The framework is based around a pipeline structure with user-replaceable signal processing modules. This structure can encapsulate a wide variety of feature extraction and classification techniques and can be applied to detecting a variety of aspects of cognitive state. Users of the framework can implement signal processing plugins in C++ or Python. The framework also provides a graphical user interface and the ability to save and load data to and from arbitrary file formats. Two small studies are reported which demonstrate the capabilities of the framework in typical applications: monitoring eye closure and detecting simulated microsleeps. While specifically designed for microsleep detection/prediction, the software framework can be just as appropriately applied to (i) other measures of cognitive state and (ii) development of biomedical instruments for multi-modal real-time physiological monitoring and event detection in intensive care, anaesthesiology, cardiology, neurosurgery, etc. The software framework has been made freely available for researchers to use and modify under an open source licence.

Keywords Biosignals · Real-time · Multi-modal · Cognitive monitoring · Software framework

✉ Simon J. Knopp
simon.knopp@nzbri.org

[1] Department of Electrical and Computer Engineering, University of Canterbury, Christchurch, New Zealand

[2] New Zealand Brain Research Institute, Christchurch, New Zealand

Introduction

Over recent years, an increasing amount of research has been undertaken in the fields of workload monitoring [1], passive brain–computer interfaces [2], and augmented cognition [3]. Techniques for detecting lapses of responsiveness (~0.5–15 s [4]), particularly microsleeps, are also being developed [4–7] towards increasing transportation safety. Many of these applications require a system that can capture multiple biosignals and process them in real time to classify various cognitive states or events.

The Elapse platform [8] was developed to be such a system, providing a common hardware and software platform to aid research in the above areas, particularly microsleep detection. It consists of two parts: a wearable device to capture biosignals and a software framework to process these signals. The device captures 16 channels of EEG, video of one eye at 60 fps, and head movement via a six-axis inertial measurement unit (IMU). The captured data is transmitted wirelessly to a remote computer running the signal processing software. This signal processing software is the focus of this paper.

Requirements

Given the characteristics of the Elapse device, several requirements exist for the signal processing software. Firstly, and most obviously, it must support at least the types of signals that the device captures: EEG, video, and inertial data. The software should provide a single integrated system for all stages of signal processing, from receiving/loading the biosignals, through feature extraction to classification and producing some output. This processing must be done in real time so that biofeedback can be provided to the user. Finally, given that Elapse is intended

to be a research platform, the software should make it easy for users to implement their own signal processing algorithms and to mix and match these at will. The software should also be able to save the raw data as it is captured and to reload this data at a later time to allow experimentation with different algorithms.

There is existing software that meets some of these goals, either as part of an integrated hardware and software solution or as standalone packages. These each have an emphasis on particular use-cases and have varying degrees of flexibility. A small sample of these systems are briefly reviewed here. Although it is by no means a complete list, it is representative of the range of what is available.

Existing platforms

The Biosignal Igniter Toolkit (BIT) [9] is a low-cost biosignal acquisition and processing system aimed at education and prototyping. The hardware portion of the system, known as BITalino [10], includes sensors for electromyography (EMG), electrocardiography (ECG), electrodermal activity (EDA), and acceleration, all connected to a Bluetooth interface. The accompanying acquisition and visualisation software, OpenSignals (previously SignalBIT [11]), receives data from the device, displays it in a graphical user interface (GUI), and saves it to disk. The stored data can then be replayed through the same software or processed with the BioSPPy toolbox. BioSPPy provides a library of common biosignal processing and feature extraction algorithms, e.g., QRS complex detection for ECG. OpenSignals does not provide the ability to process incoming data using the BioSPPy functions during real-time operation, so processing must be done offline. For its stated purpose of education, this is an acceptable model, but it does not meet our requirement for real-time processing.

BiosignalsStudio [12] is a software framework for real-time acquisition and processing of biosignals. It receives data from some acquisition device, passes it through an arbitrary combination of signal processing and feature extraction modules, and sends the output to a display, a file, and/or external classification software. The software allows the user to construct any number of parallel streams, each reading data from one sensor, applying any number of operations to the data (e.g., filtering, format conversion), and sending it to any number of outputs. This is a very flexible structure; for instance, input modules can be implemented to read data from directly connected sensors, from a wireless connection to a remote device, or from a file. The disadvantage of this system when implementing a complete biosignal-based application is that it does not include anything to do with classification. It is up to the user to implement a separate program for classification along with any necessary code to parse the output of BiosignalsStudio, and to take care of launching both programs and setting up the communication between them at run-time. This fails our requirement for a single integrated system.

G.tec, a manufacturer of EEG acquisition systems, offers software called g.HIsys [13] which provides a real-time interface between their devices and the Simulink and LabView graphical programming environments. This makes the existing libraries of Simulink and LabView signal processing blocks available for use. They also produce g.RTanalyze, a library of Simulink blocks for biosignal processing, including filtering, power spectrum analysis, heart rate variability, linear discriminant analysis (LDA), and support vector machines (SVM). While the concept of integrating biosignal acquisition hardware with one of these platforms is useful, this particular software is, of course, only useful with g.tec's hardware.

It may be feasible to apply existing multimedia processing software to this problem domain. GStreamer [14], for example, is a library which allows the user to construct pipelines of media processing elements. In a typical video player application, this might consist of a file source element followed by a demultiplexer to extract audio and video streams, decoder elements to decompress the byte streams, a video sink to display the video on screen, and an audio sink to play the audio through speakers. GStreamer can pass almost any type of data through the pipeline, representing everything simply as "buffers" with some associated metadata, so it could be equally applicable to biosignals as to audio/video. GStreamer also takes care of synchronising multiple data streams by the timestamp on each buffer. In practice, though, it is a reasonably complex framework and the effort required to conform to its API does not meet our requirement for users to be able to easily write their own plugins.

All of these existing systems meet some of the requirements of "Requirements" section but none of them meet all of the requirements. The Elapse framework uses some of the concepts from these systems and builds them into a complete framework for biosignal classification applications.

Materials and methods

At the centre of the Elapse software framework is a configurable signal processing pipeline. The pipeline has five stages, illustrated in Fig. 1. This structure was designed to be as simple as possible while still providing the flexibility necessary to encapsulate a wide variety of signal processing and classification algorithms for a variety of applications.

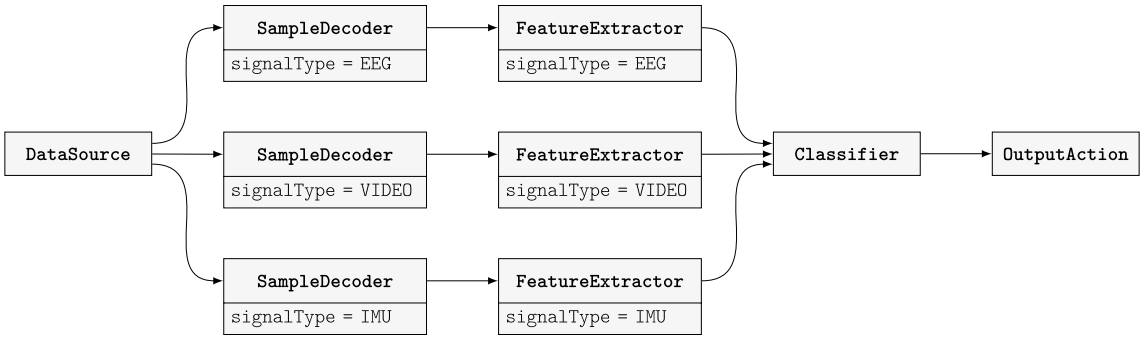

Fig. 1 Object diagram of the Elapse signal processing pipeline. *Arrows* represent data flow between objects

Signal processing pipeline

First, there is a *data source* which is responsible for pushing data into the pipeline. This is typically done by receiving data over Wi-Fi as it is captured in real time by the Elapse device, although it could also load data from file. The data source produces multiple outputs, one per biosignal type.

The next stage of the pipeline is a set of *sample decoders*, one per signal type. Each sample decoder receives one byte stream from the data source and decodes it to produce meaningful data structures. For example, the video decoder's task is to decode the H.264-compressed video stream from the device to produce a sequence of uncompressed images.

The output of each sample decoder is passed to a *feature extractor*—again, one for each signal type. The role of each feature extractor is to extract salient features from a sequence of samples. For example, an EEG feature extractor could calculate the power spectral density in a window of samples.

The fourth stage of the pipeline is the *classifier*. The classifier analyses the output of all of the feature extractors to identify the cognitive state of interest for the particular application. For the example of alertness monitoring, the output of the classifier could be whether the user is currently having a microsleep and how likely they are to have one in the next 5 min.

Finally, this information is passed to an *output action* which can take some action based on the classified state. Keeping with the example of alertness monitoring, the output action could sound an alarm to rouse the user or trigger some safety mechanism.

Pipeline elements are loosely coupled and do not interact directly with each other; all interactions between elements are mediated by a *pipeline* object. This is achieved by the use of Qt's "signals and slots" mechanism. Qt [15] is a set of C++libraries for application development, including a GUI toolkit, support for dynamically loadable plugins, and high-level networking classes. Signals and slots are essentially the observer pattern [16]—a signal is an observable event and a slot is an event handler. Connections from signals to slots can be managed at run-time and whenever a signal is emitted all connected slots are executed. Each of the pipeline elements implements one input slot and one output signal, with the exception of the data source, which only has output signals, and the output action, which only has an input slot. The signal-to-slot connections between elements are made indirectly by enqueueing the signals in Qt's event loop. This allows elements to use background worker threads internally to effectively exploit multi-core processors, while transparently ensuring that the input slot to the next element is always called from the main thread.

In parallel with the pipeline, a *data sink* observes the connections between all of the elements. The data sink is able to save any of the data passing between elements to disk in arbitrary formats.

Plugin system

All of the elements in the signal processing pipeline are provided by plugins. Plugins are discovered dynamically at run-time and the user is able to select which ones to use from a graphical dialog box. All of the elements in the pipeline can be replaced by custom versions implemented by the user, though defaults are provided. When using the software with the Elapse device there is no need to provide custom implementations for the first two stages of the pipeline—the data source and sample decoders—because the default implementation of these is tied to the device's embedded software. It is up to the user to provide meaningful implementations for the elements in the final three stages.

It is this plugin system in combination with the pipeline structure that makes the Elapse software a "framework". To quote Pree [17, p. 152]:

Application frameworks consist of ready-to-use and semi-finished building blocks. The overall architecture is predefined as well. Producing specific applications usually means to adjust building blocks to specific needs by overriding some methods in subclasses.

The Elapse software provides the overall pipeline architecture and allows the user to load signal processing "blocks" from plugins to meet the needs of their application. Currently, plugin hosts have been implemented for C++ and Python and support for Matlab is planned.

The plugin system has been designed to allow users to write plugins in multiple languages. Figure 2 shows the internal structure of the plugin management code. The *plugin manager* uses the abstract factory pattern [16] to create instances of classes provided by plugins. It contains a set of *plugin hosts*, each of which is capable of loading plugins implemented in one particular language. For example, the `NativePluginHost` loads C++ shared libraries and instantiates C++ classes, while the `PythonPluginHost` starts a Python interpreter and imports Python modules. Each plugin contains one or more implementations of the element base classes (e.g., `FeatureExtractor`, `Classifier`) along with some identifying metadata. For C++, the classes inherit directly from the Elapse base classes, but other languages require appropriate bindings (glue code) to translate to and from C++. To implement support for a new language, it is only necessary to provide a plugin host and bindings for the Elapse base classes, as the shaded region in Fig. 2 illustrates for Python.

By providing multiple interfaces to the Elapse framework, it is possible for users to choose the most appropriate language for their application. If they want to interact with existing C/C++ libraries or require low-level control over memory management, they can implement their plugins in C++. If not, they can implement their plugins in Python and take advantage of the extensive collection of mathematical and signal processing functions provided by the NumPy and SciPy libraries [18], among others.

Methods

The following section describes each element of the signal processing pipeline in greater detail, including the interfaces between elements and the data types passed through the pipeline. Also, since the Elapse framework aims to provide everything except the actual signal processing, a set of default elements are provided which are described below.

The task of the data source is to receive or load data from some source and to pass that data to the rest of the pipeline via its output signals. The default data source receives data from the device over Wi-Fi. Whenever data is received the data source simply emits the received byte array via the output signal corresponding to the signal type, e.g., `eegReady(bytes)`. The data source has no knowledge of the meaning of the data that it receives; it exists solely to move blocks of bytes around.

The sample decoders take these blocks of data and produce meaningful samples. There is a sample decoder for each biosignal. All sample decoders emit subclasses of `Sample` (Fig. 3). `Samples` have a 64-bit timestamp containing the time at which the sample was captured in nanoseconds, generated by the data capture hardware on the device. Each biosignal has a corresponding subclass of Sample. An `EegSample` contains an array of floating-point values, each representing one channel of EEG in microvolts, as well as a sequence number which is added by the EEG hardware driver for detecting dropped samples.

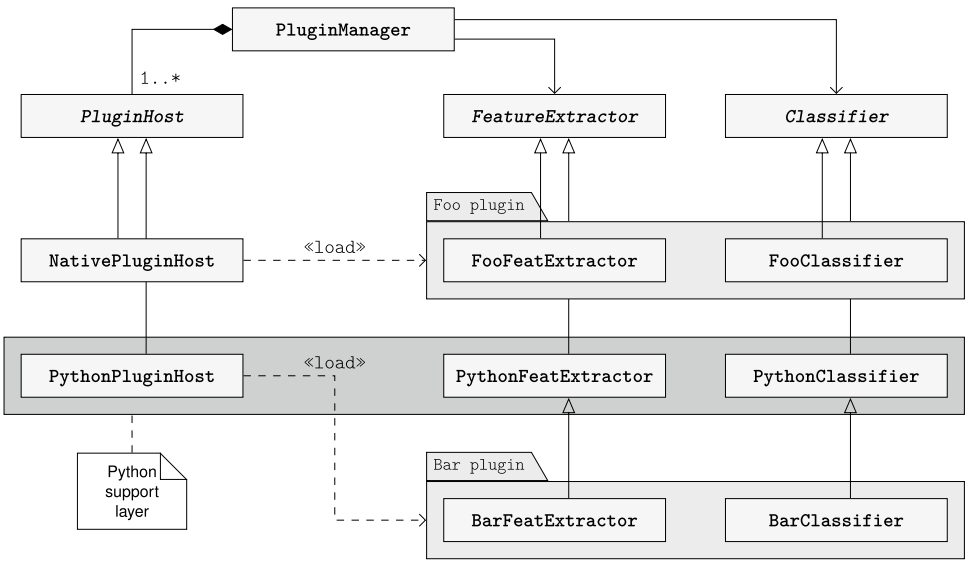

Fig. 2 Class diagram of the Elapse plugin manager. PluginHost is an abstract factory that instantiates classes provided by plugins. "Foo plugin" is implemented in C++ and "Bar plugin" in Python

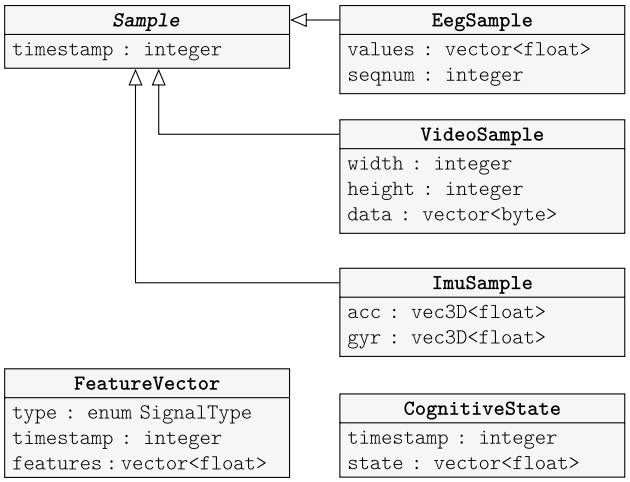

Fig. 3 Class diagram for the types passed through the pipeline

A `VideoSample` contains one frame of video as an 8-bit greyscale image. An `ImuSample` contains two three-axis values, one containing acceleration in m/s^2 and the other angular velocity in °/s. While it is possible for the user to provide their own sample decoders in a plugin, there is no need for them to do so—the default decoders have been implemented to match their encoding counterparts in the device's embedded software.

Unlike the data source and sample decoders, users must implement feature extractors specific to their application. Each feature extractor receives the samples emitted from a sample decoder and extracts features that are important for the particular application. From the user's perspective a feature extractor has two properties, T_{win} and T_{step}, and two methods, `analyseSample()` and `analyseWindow()`. The window length T_{win} and step size T_{step} define a sliding window of time; these values are the same for all feature extractors in the pipeline. The two methods allow the user to extract features both from individual samples and from a window of samples. For example, an implementation of a video feature extractor could implement an `analyseSample()` method that analyses a frame of video to locate the pupil and eyelids. An EEG feature extractor, on the other hand, would not need to implement this method since there are no features that can be extracted from a single sample of EEG. The feature extractor base class takes the result of the `analyseSample()` method and pushes it into an internal queue. Once a full window of samples has been received, the contents of this queue are passed to the `analyseWindow()` method. Users can implement this method to identify temporal features in the queued data. To continue with the earlier examples, the video feature extractor implementation could analyse the sequence of pupil and eyelid positions to count blinks and calculate the percentage of time that the eyes were closed. The EEG feature extractor could analyse the sequence of samples to calculate the power spectral density in each channel. The output of the `analyseWindow()` method is a `FeatureVector`—an array of floating-point values with the exact number and meaning of each left up to the person implementing the feature extractor. Once the feature vector has been emitted into the pipeline, the feature extractor base class removes all data within T_{step} of the oldest sample in the internal queue, thus sliding the window to its next starting point. This feature extractor design hides all of the details of dealing with sample timestamps and windowing logic from the user, allowing them to simply implement one or two functions to do the actual signal processing.

Like the feature extractors, the user must implement a classifier specific to their application. From the user's perspective a classifier has a single method: `classify()`. This method is passed a set of feature vectors for one window of time, one for each signal type. The user must override this method to implement a multimodal classifier which analyses the set of feature vectors to identify the cognitive state of the subject. This state may be almost anything according to the requirements of the application—attention, arousal, alertness, anxiety, drowsiness, fatigue, vigilance, workload, boredom, excitement, etc. The output of the `classify()` method is a `CognitiveState` object which contains an array of floating point values. Much like for `FeatureVectors`, the number and meaning of these values is left up to the user. Behind the scenes, the classifier base class takes care of synchronising the feature vector inputs. The output signals of all of the feature extractors are connected to the classifier's one input slot; the base class buffers the incoming feature vectors, inspects their timestamps, and calls the `classify()` method when the full set of feature vectors is available for one time window. This technique accounts for any differences in processing time required by the sample decoders and feature extractors for the different signal types. Once again, this hides the complexity of dealing with timing issues from the user and allows them to implement just their classification algorithm.

The user may implement an output action to take some action based on the classified cognitive state. While the Elapse device is the interface from the real world to the analysis software, the output action is the interface from the software back to the real world. It is what makes a system based on this framework of practical benefit. Returning to the example of alertness monitoring for transportation, an output action could sound an alarm to alert the subject, or even trigger an adaptive cruise control system to prepare for emergency braking. Similarly, in a clinical environment a multi-modal physiological instrument could sound an alarm if abnormal events or state were detected.

Finally, a default data sink is provided which saves each byte array emitted by the data source to disk along with the timestamp at which it was received. A corresponding data source element is also provided which can load the files saved by the data sink and, using the stored timestamps, "replay" the data through the pipeline again at the same rate. This is useful in a research context so that all of the raw data can be saved during a study. When it is later reloaded by the data source, it is exactly as if the device was operating in real time and so can be used for testing feature extractors and classifiers.

Users may implement their own data sinks to save data to any required format. The data sink observes the connections between all of the elements so it would be possible to create a sink that, for example, saves the decoded EEG to a Matlab file, the video frames to a video file, and the classifier output to a text file. Data sinks also support the concept of *capture info*—arbitrary metadata associated with a captured data set. This could be as simple as the time and date, or as complex as the subject ID and test conditions for a research study. The user implementing the data sink is free to choose what capture info is required, if any, and how to save that information to disk.

User interface

The signal processing pipeline is wrapped up in a user-friendly application to allow easy interaction with the software. In addition to the signal processing pipeline the application contains a GUI, support for controlling the device hardware, support for configuration files, and the plugin management code (see "Plugin system" section).

The GUI (Fig. 4) has two main purposes. The first is to allow the user to select which element classes to load from plugins to populate the pipeline. When the application is first launched it searches for available plugins and presents the user with a window to choose which elements to use. The user must select one implementation for each element in the pipeline. This selection is saved in the configuration file and will be reloaded automatically in subsequent sessions.

The second purpose of the GUI is to monitor the operation of the signal processing pipeline as it runs. Elements may optionally implement a *displayable* interface which allows them to provide a GUI widget to the application. For example, the default EEG sample decoder provides a strip-chart that plots the last several seconds of EEG data and the IMU decoder displays a 3D head in the same orientation as the subject's head. User-implemented pipeline elements are free to implement the displayable interface to provide a graphical representation of their internal state. For example, in Fig. 4 the eye video feature extractor that is being used provides a widget which displays the pupil boundary overlaid on the eye video. All of the widgets provided by displayable elements are shown in the main window of the application. These widgets can be rearranged within the window to suit the user's preference.

The application also has a log window which displays messages from the code. The list of messages can be filtered according to severity (e.g., debug, warning, error) and searched using regular expressions. This centralised logging facility collects messages from the application itself, from the default elements, and from user-implemented elements. This is very useful for debugging.

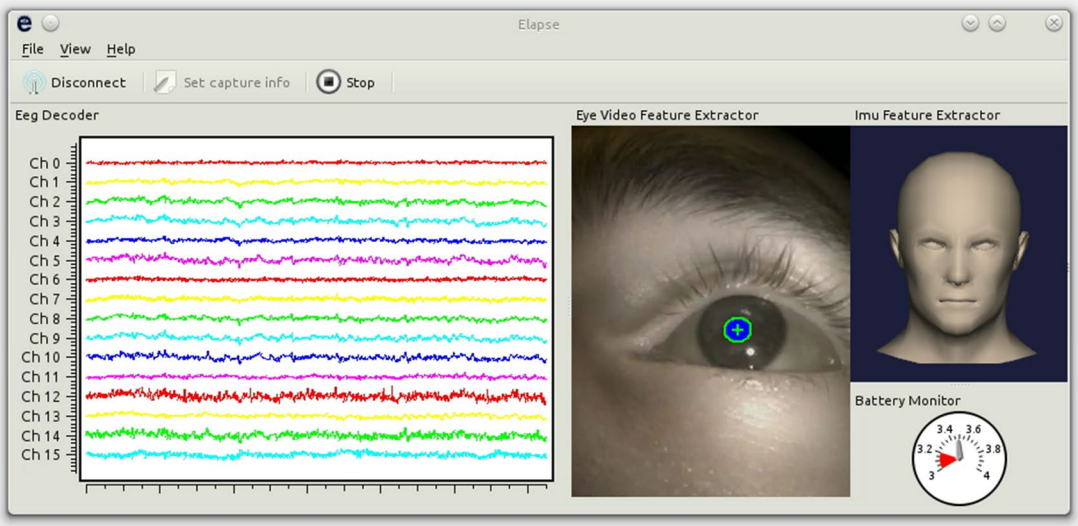

Fig. 4 The Elapse framework's user interface during data capture

Results

To demonstrate the operation of the software, signal processing plugins were implemented to detect simulated microsleep events. An experiment was carried out using these elements to demonstrate that it is possible to extract meaningful information from the captured data and process it through the Elapse framework.

The experiment was conducted with a single subject. The subject performed two tasks: the first, the 'closed-eyes task', was performed by looking at a fixed point in front of him while keeping his eyes open for 15 s, followed by closing his eyes for 15 s when prompted by an audible beep. This sequence was repeated three times. This task was designed to confirm that the software could detect changes in EEG spectral content during eye closure. In the second task, the 'simulated sleep' task, the subject performed exaggerated simulated microsleeps—closing his eyes slowly while drooping his head forwards, then quickly opening his eyes and jerking his head back upright. This was done three times, interspersed with 15 s periods of maintaining gaze on a fixed point. This second task was designed to cause measurable changes in all three of the signals—eye closure in the eye video, increased posterior alpha in the EEG, and forward head tilt from the IMU. Both tasks were done with the subject wearing the Elapse device (Fig. 5). The software pipeline was configured with the feature extractors and classifiers described in "Simulated sleep task" section as well as a data sink that saved all of the raw data to disk.

Fig. 5 A user wearing the prototype Elapse device

Closed-eyes task

In the majority of people (~80%) an increase in posterior alpha activity occurs during restful wakefulness with the eyes closed compared to with eyes open [19]. Figure 6 shows a spectrogram of the EEG captured at O1 during 2.5 periods of alternating eyes open and closed. The increase in alpha-band power (8–12 Hz) is clearly visible during the two eyes-closed periods. Figure 7 shows the same data in the time domain, bandpass filtered between 4 and 40 Hz, as two 10 s windows centred at the onset of each of the periods of eye closure in Fig. 6. Again, the increase in alpha activity is clearly visible when the eyes are closed at 15 and 45 s.

Simulated sleep task

The signal processing elements implemented for the second task were deliberately simplistic, keeping the focus on the operation of the system as a whole rather than on advanced signal processing techniques. The task produces easily measurable changes in the signals so there is no need for complicated processing. The window size of the feature

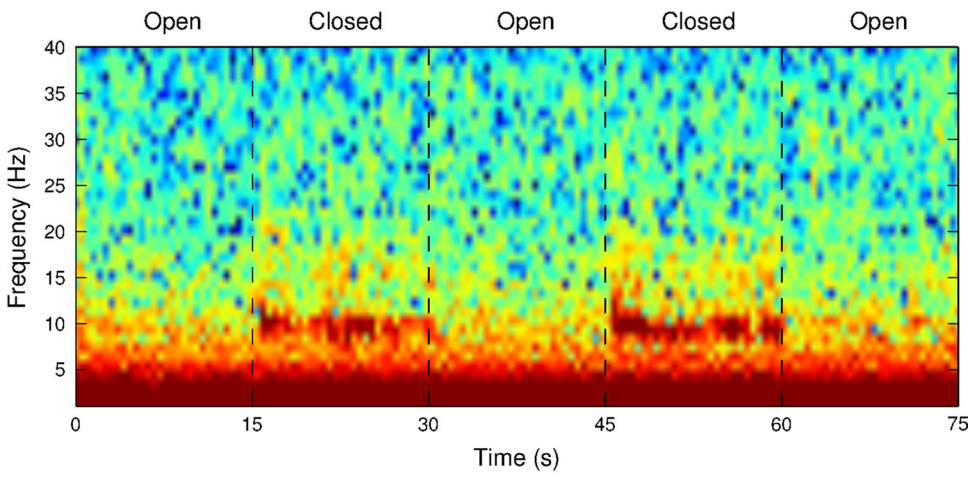

Fig. 6 Spectrogram of occipital EEG (O1) showing increased alpha during eye closure

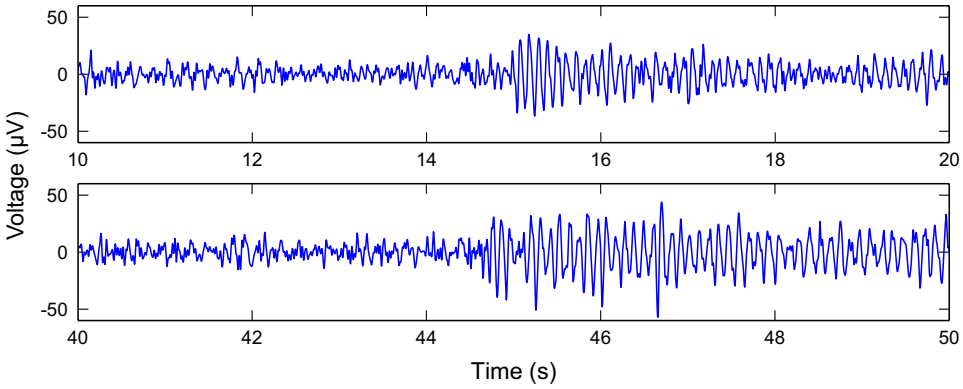

Fig. 7 Increase in occipital alpha during eye closure. The two traces show 10 s windows around the beginning of the periods of eye closure shown in Fig. 6

extractors was set to $T_{\text{win}} = 2$ s with a step size of $T_{\text{step}} = 0.5$ s.

The eye video feature extractor used the algorithm from [20] to locate the boundary of the pupil in each frame of video and to assign one of three categories of eye closure: open, partially closed, or closed. The eye closure categories were used to calculate the percentage of time in each 2 s window in which the eye was closed, denoted here as PERCLOS_2. (Note that this is different to the usual definition of PERCLOS which measures percentage eye closure during a 1-min window [21].) The EEG feature extractor used the squared magnitude of the Fourier transform of the data to approximate the power spectral density. It used this to calculate the total power between 4 and 40 Hz in EEG channel O1. The IMU feature extractor calculated the cumulative change in the head pitch (nod) angle in each 2 s window.

The classifier simply applied a threshold to each of the features (PERCLOS_2, EEG power, head pitch) and considered a simulated microsleep event to have occurred when all of the three features exceeded their respective thresholds. This does not reflect any of the subtleties of detecting real microsleeps but it is sufficient to demonstrate the process of classifying an event based on features from multiple modalities.

The upper three traces of Fig. 8 show the variation of the three extracted features over the course of three simulated microsleeps. The dotted lines mark the threshold that the classifier applied to each feature. The output of the classifier is shown in the bottom trace, reading 1 when all of the features exceed their thresholds and 0 otherwise. The plot clearly shows that the classifier successfully detects the three simulated microsleep events.

More importantly, however, this example application demonstrates the successful operation of an end-to-end biosignal classification system. It loads signal processing code from user-implemented plugins written in both C++ and Python; it receives biosignal data in real time; it demonstrates the two aspects of feature extraction (per-sample analysis for the video and per-window analysis for all signals); it demonstrates a multi-modal classifier, and it presents all of this to the user as a single graphical application. None of the code loaded from plugins needed

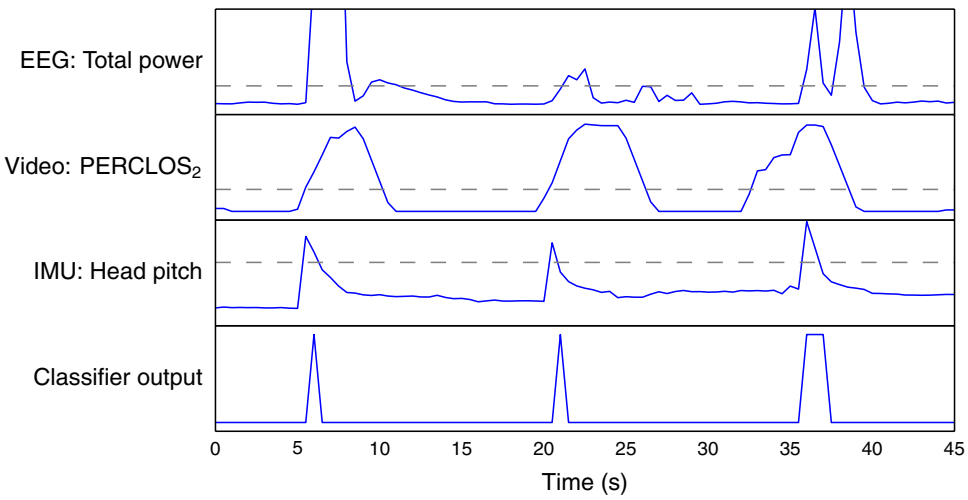

Fig. 8 Detecting simulated microsleeps. The top three traces show features extracted from the biosignals. The classifier identifies simulated microsleeps when all three features are above certain thresholds (*dashed lines*)

to deal with timestamps and synchronisation because the pipeline handles all of that.

Software characteristics and performance

Latency is an important factor in all real-time systems. To measure the latency of the Elapse system, a circuit was devised that generated an event visible to all of the device's sensors at the same time: an LED positioned in front of the camera lit up, a voltage pulse was applied between two EEG electrodes, and a servo rotated to tap the IMU [8]. The latency for each sensor was defined as the time between the event being triggered and the sample containing that event being emitted by the corresponding sample decoder in the Elapse software. That is, it is the total round-trip time for the laptop to trigger the event, the sensor to sample the event, the device's firmware to capture and transmit the sample containing the event, and the Elapse software to receive and decode that sample. It does not include the time taken to process the sample through the feature extractors and classifier since that is entirely dependent on the user's implementation of those elements. Twenty of these events were recorded and the latencies for all three channels were measured to be less than 100 ms. A microsleep detection device needs to detect and respond to events lasting more than 500 ms [4], so a latency of 100 ms allows detection soon after the onset of a microsleep.

Significant effort has been put into making it easy for users to implement their own algorithms within the framework and also to expand the framework as necessary. The most common requirement that users will have, and therefore the one that has been made easiest to achieve, is to implement their own feature extraction and classification algorithms. This is simply a matter of copying a small amount of template code from the sample provided to create a new plugin, then filling in the blanks. The same applies for implementing custom data sinks/sources and output actions and for displaying custom GUI widgets. If the user is capturing data with hardware other than the Elapse device they may want to add support for other types of signals. This involves modifying the framework but is fairly simple to achieve by adding a field to the signal type enumeration and a corresponding branch to the pipeline. If users have existing signal processing code in a language that is not supported by the framework then they may wish to implement a new plugin host to support plugins written in that language. Plugin hosts have a very simple factory interface so the difficulty of implementing one depends mostly on how easy it is to call into that language from C++.

Discussion

The Elapse software framework as it is described above provides a combination of features that is not present in the existing software mentioned in "Existing platforms" section. It processes signals in real time, unlike the Biosignal Igniter Toolkit [9]. It encapsulates the whole signal processing pipeline from data capture to classification, unlike BiosignalsStudio [12]. It is designed specifically for processing biosignals, unlike GStreamer [14].

Although the pipeline structure supports a variety of signal processing and classification techniques, there are some things that its simple structure does not allow. For example, it is not possible to implement a feature extractor for one signal that uses other signals to improve its performance, e.g., using the eye video to help remove eye-blink artefacts from the EEG. The classifier, however, uses the features from all of the signals, so this could potentially be implemented as a preprocessing step inside the classifier.

An important factor driving the design of the Elapse software was the high data rate of the eye video—the uncompressed video has a bit rate of 37 Mbit/s. Because this rate is much higher than the more commonly acquired biosignals such as EEG and EMG, the software framework has constraints that are not present in some other biosignal acquisition systems. For example, at high data rates it is useful to have control over memory management in order to avoid copying large blocks of data unnecessarily. The Elapse framework uses reference-counting "smart pointers" to manage the lifetime of the data passed through the pipeline. That is, space for the data is allocated dynamically and that space is automatically freed when nothing refers to the data any more. It is possible for users to implement more efficient memory management techniques when writing plugins, too. For example, a video decoder element could allocate frames of video data from an internal buffer pool [22] and provide a custom destructor for the smart pointer which returns the buffers to the pool when they are no longer needed. This technique is faster than allocating every new frame of video dynamically. To the rest of the Elapse framework, nothing has changed because the data is still passed around using the same type of smart pointer. This gives the user flexibility to implement any custom memory management techniques that may be necessary for their application.

The Elapse device and software framework have been developed as part of a larger project to detect and predict microsleeps. In this context, future work will likely involve porting some of the techniques that have been developed to run within the framework. These include using various EEG features (power spectrum, approximate entropy, fractal dimension, Lempel–Ziv complexity) with various classifiers (tapped delay-line multilayer perceptron and long

short-term memory recurrent neural networks, echo state networks, linear discriminant analysis) [5, 6, 23, 24]. To apply the system to real-world microsleep detection, it will also be necessary to run experiments with more subjects and to analyse real microsleeps, since the proof-of-concept experiments described here used a single subject and simulated microsleeps.

While the device and software framework were specifically designed for microsleep detection and prediction, they can equally be used to quantify other measures of cognitive state, and to develop biomedical instruments for multi-modal real-time physiological monitoring and event detection in intensive care, anaesthesiology, cardiology, neurosurgery, etc.

Conclusion

We have developed a software framework for implementing real-time, multi-modal cognitive monitoring applications. It provides a pipeline structure which allows users to load custom signal processing code from plugins written in C++ or Python. The pipeline handles synchronisation between signals so that the user does not have to. The framework also provides facilities for saving, loading, and visualising data in real time. This combination of features was not available in existing software. Preliminary studies have demonstrated the usefulness of the framework in supporting signal processing for research into cognitive states.

The Elapse software has been released under an open-source licence and is available from http://github.com/sijk/elapse. This licence allows anyone to use or contribute to the core framework while also allowing signal processing plugins to remain closed-source if necessary.

Acknowledgements Simon Knopp was the recipient of a University of Canterbury Doctoral Scholarship and the work reported formed part of his doctoral study.

Compliance with ethical standards

Conflict of interest The authors declare that they have no financial or personal relationships with other people or organisations that could have inappropriately influenced this work.

Ethical approval Ethical approval was not required due to the small scale and non-invasive nature of the experiments.

References

1. Borghini G, Astolfi L, Vecchiato G, Mattia D, Babiloni F (2014) Measuring neurophysiological signals in aircraft pilots and car drivers for the assessment of mental workload, fatigue and drowsiness. Neurosci Biobehav Rev 44:58–75. doi:10.1016/j.neubiorev.2012.10.003
2. Zander TO, Kothe C (2011) Towards passive brain–computer interfaces: applying brain–computer interface technology to human–machine systems in general. J Neural Eng 8:25005. doi:10.1088/1741-2560/8/2/025005
3. Reeves LM, Schmorrow DD, Stanney KM (2007) Augmented cognition and cognitive state assessment technology – near-term, mid-term, and long-term research objectives. In: Foundations of augmented cognition. Springer, Berlin, pp. 220–228. doi:10.1007/978-3-540-73216-7_25
4. Jones RD, Poudel GR, Innes CRH, Davidson PR, Peiris MTR, Malla AM, Signal TL, Carroll GJ, Watts R, Bones PJ (2010) Lapses of responsiveness: characteristics, detection, and underlying mechanisms. In: Proceedings of 32nd IEEE Conference on Engineering in Medicine and Biology Society. pp 1788–1791. doi:10.1109/IEMBS.2010.5626385
5. Davidson PR, Jones RD, Peiris MTR (2007) EEG-based lapse detection with high temporal resolution. IEEE Trans Biomed Eng 54:832–839. doi:10.1109/TBME.2007.893452
6. Peiris MTR, Davidson PR, Bones PJ, Jones RD (2011) Detection of lapses in responsiveness from the EEG. J Neural Eng 8:16003. doi:10.1088/1741-2560/8/1/016003
7. Golz M, Sommer D, Chen M, Trutschel U, Mandic D (2007) Feature fusion for the detection of microsleep events. J VLSI Signal Process 49:329–342. doi:10.1007/s11265-007-0083-4
8. Knopp SJ (2015) A multi-modal device for application in microsleep detection. PhD thesis, University of Canterbury. http://hdl.handle.net/10092/10408. Accessed 04 Oct 2015
9. da Silva HP, Lourenço A, Fred A, Martins R (2014) BIT: biosignal igniter toolkit. Comput Methods Progr Biomed 115:20–32. doi:10.1016/j.cmpb.2014.03.002
10. BITalino DIY biosignals. http://bitalino.com/. Accessed 04 Oct 2015
11. Lourenço A, da Silva HP, Carreiras C, Alves AP, Fred A (2014) A web-based platform for biosignal visualization and annotation. Multimed Tools Appl 70:433–460. doi:10.1007/s11042-013-1397-9
12. Heger D, Putze F, Amma C, Wand M, Plotkin I, Wielatt T, Schultz T (2010) BiosignalsStudio: a flexible framework for biosignal capturing and processing. In: Dillmann R, Beyerer J, Hanebeck UD, Schultz T (eds) Annual conference on artificial intelligence. Springer, Berlin/Heidelberg, pp 33–39. doi:10.1007/978-3-642-16111-7_3
13. Guger Technologies. http://www.gtec.at/Products/. Accessed 04 Oct 2015
14. GStreamer: open-source multimedia framework. http://gstreamer.freedesktop.org/. Accessed 04 Oct 2015
15. Qt. http://qt.io/. Accessed 04 Oct 2015
16. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Pearson Education, Upper Saddle River
17. Pree W (1994) Meta patterns—a Means for capturing the essentials of reusable object-oriented design. In: Object-oriented program. Springer, Berlin, pp 150–162
18. Jones E, Oliphant T, Peterson P et al (2001) SciPy: open source scientific tools for Python. http://www.scipy.org/. Accessed 04 Oct 2015
19. Santamaria J, Chiappa KH (1987) The EEG of drowsiness in normal adults. J Clin Neurophysiol 4:327–382. doi:10.1097/00004691-198710000-00002
20. Knopp SJ, Bones PJ, Weddell SJ, Innes CRH, Jones RD (2013) A wearable device for measuring eye dynamics in real-world conditions. In: Proceedings of 35th IEEE Conference on Engineering in Medicine and Biology Society, pp 6615–6618. doi:10.1109/EMBC.2013.6611072

21. Wierwille WW, Ellsworth LA (1994) Evaluation of driver drowsiness by trained raters. Accid Anal Prev 26:571–581. doi:10.1016/0001-4575(94)90019-1
22. Boulton RJ, Walthinsen E, Baker S, Johnson L, Bultje RS, Kost S, Müller T-P, Taymans W (2015) GStreamer plugin writer's guide, Ch 15: memory management. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/chapter-allocation.html. Accessed 04 Oct 2015
23. Ayyagari SSDP, Jones RD, Weddell S (2015). Optimized echo state networks with leaky integrator neurons for EEG-based microsleep detection. In: Proceedings of 37th IEEE Conference on Engineering in Medicine and Biology Society, pp 3775–3778. doi:10.1109/EMBC.2015.7319215
24. Shoorangiz R, Weddell S, Jones RD (2016). Prediction of microsleeps from EEG: preliminary results. In: Proceedings of 38th IEEE Conference on Engineering in Medicine and Biology Society, pp 4650–4653