UsyBus: A Communication Framework among Reusable Agents integrating Eye-Tracking in Interactive Applications

FRANCIS JAMBON, Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
JEAN VANDERDONCKT, Université catholique de Louvain, LouRIM, Belgium

Eye movement analysis is a popular method to evaluate whether a user interface meets the users' requirements and abilities. However, with current tools, setting up a usability evaluation with an eye-tracker is resource-consuming, since the areas of interest are defined manually, exhaustively and redefined each time the user interface changes. This process is also error-prone, since eye movement data must be finely synchronised with user interface changes. These issues become more serious when the user interface layout changes dynamically in response to user actions. In addition, current tools do not allow easy integration into interactive applications, and opportunistic code must be written to link these tools to user interfaces. To address these shortcomings and to leverage the capabilities of eye-tracking, we present UsyBus, a communication framework for autonomous, tight coupling among reusable agents. These agents are responsible for collecting data from eye-trackers, analyzing eye movements, and managing communication with other modules of an interactive application. UsyBus allows multiple heterogeneous eye-trackers as input, provides multiple configurable outputs depending on the data to be exploited. Modules exchange data based on the UsyBus communication framework, thus creating a customizable multi-agent architecture. UsyBus application domains range from usability evaluation to gaze interaction applications design. Two case studies, composed of reusable modules from our portfolio, exemplify the implementation of the UsyBus framework.

CCS Concepts: • **Software and its engineering** → *Development frameworks and environments*; *Software libraries and repositories*; • **Human-centered computing** → *Usability testing*; *Systems and tools for interaction design.*

Additional Key Words and Phrases: Communication data bus, Eye movement analysis, Oculometry, Eye-tracker, Eye-tracking, Usability engineering

ACM Reference Format:
Francis Jambon and Jean Vanderdonckt. 2022. UsyBus: A Communication Framework among Reusable Agents integrating Eye-Tracking in Interactive Applications. *Proc. ACM Hum.-Comput. Interact.* 6, EICS, Article 157 (June 2022), 36 pages. https://doi.org/10.1145/3532207

1 INTRODUCTION

In Human-Computer Interaction (HCI) as well as in Information Visualization (IV), eye-tracking studies [7] are particularly effective to demonstrate the benefits of some design alternatives for a particular user interface, to prove that a redesign improved the usability (*e.g.*, in terms of efficiency of visual scan paths), or to analyze how a Graphical User Interface (GUI) can be customized to a

Authors' addresses: Francis Jambon, Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, Bâtiment IMAG, 700 av. centrale, 38000 Grenoble, France, francis.jambon@imag.fr; Jean Vanderdonckt, Université catholique de Louvain, LouRIM, Place des Doyens, 1, Louvain-la-Neuve, 1348, Belgium, jean.vanderdonckt@uclouvain.be.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2573-0142/2022/6-ART157 $15.00
https://doi.org/10.1145/3532207

particular audience [27, 28] and involve many information visualization techniques [6]. Yet, such eye-tracking studies specifically face major challenges such as, but not limited to [10, 14, 17]:

- **A GUI is dynamic**: as opposed to fixed documents and stimuli with a static layout and structure, a GUI contains variable elements, such as interactive widgets (*e.g.*, a drop-down list) or animated elements (*e.g.*, a banner that changes its content). More generally, a GUI dynamically changes according to end-users' actions or system responses (*e.g.*, scrolling, data input/output, message display). All these aspects make the GUI largely evolving over time, which requires proper identification of eye movements targets.
- **A GUI is made of many elements**: as opposed to other stimuli like an advertisement or a document with limited regions, a GUI is made up of elements that are clearly identifiable to become areas of interest, but which are numerous and tedious to define. For example, defining an area of interest for each widget, each group of widgets or each container is a very repetitive and time-consuming task.
- **A GUI eye-tracking study is repeated several times**: whether it is for comparative testing, A/B testing [41], optimization or GUI adaptation [33], all steps of the eye-tracking study need to be repeated each time a new version of the GUI is designed, which renders the process even more resource-consuming and error-prone. Moreover, for GUI adaptation, a new study should be conducted each time the GUI is adapted, which may require a new scene definition, even possibly at run-time.
- **A GUI eye-tracking study produces low-level raw data**: a study produces generic data, such as gaze points, fixations, and saccades. These data are not specifically tailored to evaluating the GUI composition, layout, and structure, thus requiring further interpretation from the experimenter. Existing software does not specifically structure and analyze this data in terms of GUI elements.

To address these challenges and to leverage the capabilities of eye-tracking in HCI, this paper presents UsyBus, a framework for supporting eye-tracking studies that is potentially applicable to any interactive application and its GUI, whatever the type is (*e.g.*, a custom-made application, a commercial software, a web site) and whatever the platform is (*e.g.*, on a desktop, on a tablet, a responsive web site running on a smartphone). UsyBus is structured according to a modular multi-agent software architecture with various agents responsible for different roles (*e.g.*, eye-tracking data acquisition, data filtering, data analysis, data monitoring, data recording) sending data to the exchange bus and/or receiving data from it, thus supporting multiple input-multiple output. All agents communicate to each other via the bus, thus ensuring autonomy of the agents with respect to each other and tight coupling. In particular, various agents are developed to acquire data from different eye-trackers, thus achieving some device independence. All agents provide Multiple Input-Multiple Output (MIMO) capabilities that are parameterizable at run time.

The contributions to Engineering Interactive Computing Systems (EICS) are as follows:

- A comparative analysis of software existing for supporting eye-tracking studies in GUI development and usability evaluation (Section 2);
- A definition and an implementation of UsyBus, a multi-agent software architecture of modules ensuring different roles towards conducting an eye-tracking study, with a formal specification of the agents and their message passing mechanism (Section 3);
- A collection of reusable UsyBus agents for eye-tracking studies (Section 4);
- An instantiation of the UsyBus architecture for two case studies, highlighting the advantages and shortcomings of the engineering approach (Section 5).

Section 6 concludes the paper and suggests some future avenues to this work.

2 RELATED WORK

This section overviews the wide range of eye-tracking studies targeted to the development of interactive applications, whether they are stand-alone or web-based. Selected software environments, addressing the needs of such studies, are discussed in a more generic way.

2.1 Eye-Tracking Studies on Development Tasks

Eye-tracking has been performed on various software artifacts, such as programming code, configurations, models to support software engineering or programming tasks, and GUI design.

For instance, Shaffer *et al.* [38] developed iTrace, an Eclipse plug-in that incorporates eye-tracking in an Integrated Development Environment (IDE) to analyze developers' eye movements while developing. With this plug-in, the researchers conducted an eye-tracking study to investigate the eye movement patterns of software engineers manipulating software artifacts (*e.g.*, Java code, text/HTML/XML documents, diagrams) throughout the development life cycle. iTrace has therefore a wide range of applications provided that the development task remains in the realm of the IDE. Since such an IDE typically generates multiple graphical and textual views of programs, these views need to be coordinated with the main program understanding by the developers. To analyse this coordination, data mining techniques detect high frequency patterns of eye movements. Different visual patterns were found among participants based on their programming experience, IDE familiarity, and debugging performance [23].

Similarly, Obaidellah *et al.* [36] examined participants' visual attention when solving algorithmic problems in the form of programming problems using Blickshift software [8]. In addition, Sharif *et al.* [39] confirmed that the longer time developers spend in initially scanning the software code, the quicker they find the defect based on scan time. Program comprehension [4] and code summarization tasks [1] can be also approached by eye movement analysis. Approaches from research on natural language text reading are also applicable to source code, however not without review [13]. Among all these tasks, scanning and visual searching are the most frequently investigated tasks. For instance, Cutrell *et al.* [15] investigated how changing the presentation of search results influences their perception: adding information to the contextual snippet improved performance for a search task, but not for a navigational task. Prior work has mainly investigated eye-tracking for simple target search tasks or for general interest, where gaze behavior is less complex to analyze than in situations where the whole gaze interaction is considered [19].

A special attention has been also paid to the Graphic User Interface (GUI) of interactive applications for design, generation, adaptation, and evaluation. For instance, Bojko [10] redesigned a web page layout resulting in fewer fixations and more dense areas in the heat maps for search tasks. Vidyapu *et al.* [42] proposed a visual feature-based attention prediction model for widgets of a web page, based on a Multi-class Support Vector Machine (MSVM), to learn the visual features and their associated attention. Mirjana *et al.* [26] exploited eye-tracking technologies to observe user behavior via their psychopathological response, to dynamically adapt the presentation of the content in an e-learning environment. Holmes & Zanker [24] captured gaze fixations on a GUI to measure its aesthetic quality. Furthermore, Cheng & Dey [14] ran an interactive genetic algorithm to compute a design fitness function based on the number of fixations, the fixation durations, and the first fixation on a target GUI element. Based on these measures, they inferred end-users' preferences and adapted the GUI. More recently, Feit *et al.* [19] relied on six metrics to identify which information is relevant for a decision making process, in order to adapt the GUI accordingly to adjust the perceived information depending on the user and his/her tasks.

In summary, the body of knowledge regarding eye movement behavior analysis at various stages of the development life cycle of a software [9], or an interactive application with its user interface, is extensive [14, 21, 27, 28] and is likely to grow.

2.2 Software Environments for Eye-Tracking

The previous subsection points out that many types of eye-tracking analyses exist in the area of software development, as well as in the more general area of interactive applications. However, most of these works, if not all, are developed specifically for their intended purpose and not for supporting three important ISO/IEC 25010 [25] software quality factors:

(1) *Compatibility:* "degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment". It also includes *interoperability*, defined as the "degree to which two or more systems, products or components can exchange information and use the information that has been exchanged". Existing projects and applications rarely communicate and exchange their data to ensure interoperability.
(2) *Maintainability:* "degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements". It also includes *reusability*, defined as the "degree to which an asset can be used in more than one system, or in building other assets" and the *modifiability*, defined as the "degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality". It is unclear how existing environments for eye-tracking could be modified to be reused in another context than the one initially planned.
(3) *Portability:* "degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another". Similarly, it is unclear whether existing works can address portability, namely by replacing an eye-tracker by another one, while minimizing re-coding. In principle, a generic environment should be independent of any particular eye-tracking technology.

Fortunately, there are some concrete initiatives that address the aforementioned characteristics and their sub-characteristics, but not exactly as we are envisioning them.

For instance, the Open Gaze and Mouse Analyzer (OGAMA) [43] consists of a software written in C#.NET for recording and analyzing of eye- and mouse-tracking data from slideshow-based experiments. It supports designing the slideshow, recording of gaze and mouse data concurrently, pre-processing and filtering these data, creating attention maps and areas-of-interest. OGAMA runs only in Windows environments and supports many commercial and open-source eye-trackers. Even very useful for classic eye-tracking experiments, OGAMA lacks from interoperability since it is not possible to link it with external systems at run-time.

EyeSite [29] is an open-source software for conducting eye-tracking studies in a web browser with real-time communication. Since a web page may require scrolling, screen-space gaze coordinates are mapped into coordinates on the web page using DOM elements. EyeSite supports a wide variety of eye-trackers, remote experimental trials, and easy integration with experimental workflows. However, EyeSite is focusing on GUIs for web sites only, and therefore lacks from reusability.

PyTrack [20] consists of a Python toolkit for analyzing eye-tracking datasets based on minimal code instrumentation: after recording eye movements, the raw data are exported from the eye-tracker and imported into PyTrack in order to perform parameter extraction, generate plots, and conduct inferential statistical analysis on the extracted parameters. When other statistical tests need to be computed, PyTrack exports these data in a compatible format, like JSON. The toolkit automatically generates and visualizes gaze plots, gaze heat maps, dynamic pupil and gaze plots, and aggregates heat maps for a group of participants. PyTrack also extracts individual parameters related to pupil size, blinks, fixations, saccades, microsaccades, and reading behavior. A particularly useful feature for GUI eye-tracking is that an Area Of Interest (AOI) can be defined to extract more advanced parameters such as the number of revisits. However, PyTrack is not intended to analyze

GUIs in particular and does not support automated definition of multiple AOIs. Instead, it is mainly oriented towards performing advanced statistical analysis on eye movement data.

PyGaze [16] consists of a software toolbox for writing eye-tracking experiments in Python syntax with programming ease and script readability, without constraining functionality and flexibility. PyGaze can be used for visual and auditory stimulus presentation, for response collection via keyboard, mouse, joysticks, or eye movements using a custom algorithm, etc. Moreover, PyGaze implements an on-line saccade detection algorithm to be used in gaze-contingent applications, and links with external modules can be made via the Python scripting mechanism. While PyGaze already offers the basic abstraction to relieve the experimenter from manually programming everything, there is still a need for Python programming for experiments with special needs. PyGaze address some of the aforementioned shortcomings, but lacks portability and interoperability since Python is the only language used, and no generic communication system is defined between modules.

On the commercial side, most oculometers manufacturers offer software that is compatible only with their own devices. However, existing commercial software provides some interesting genericity, such as BlickShift[1]. Compatible with many existing eye-tracking devices, BlickShift is a software for recording and analyzing eye-tracking data with advanced visual analytics, thus greatly facilitating the handling of eye-tracking raw data and their interpretation. For example, information visualization techniques could exploit the AOI hierarchy to analyze fixation sequences on different levels of aggregation, such as AOI View, AOI Tree, AOI Matrix, and AOI Graph [8]. However, BlickShift is first and foremost an analysis software, and can not be linked to any other application.

EyeWorks[2] from EyeTracking is also compatible with eye-trackers from EyeTech, Tobii Pro, and LC Technologies, thus supporting portability. EyeWorks enables the experimenter to design a 3D scene environment, to configure the experimental test script, to calibrate the oculometer, to record data, and to analyze them afterwards. While interesting for a wide range of experiments, the EyeWorks environment turns out to be tedious for studies involving human-computer interaction, such as for interactive software: data and video sessions are recorded in real time, but can be analyzed only post-experiment, thus requiring to modify the setup each time a new interaction needs to be studied. Each area of interest needs to be defined manually after the session ended, which should be repeated for each GUI widget or element. In addition, these statically defined zones are no appropriate for dynamic user interfaces, whose contents change at run-time.

Whereas all these approaches are recognized for delivering an added value to usability evaluation and/or gaze interaction, it remains unclear how eye-tracking can be more generally incorporated into a software development life cycle to achieve its full potential [3]. Limits focus in particular on the ability to track and analyze gaze in real time on a GUI that evolves in an unplanned or unpredictable way. Yet, these software are too often tied to a particular model or family of eye-trackers, to a particular type of application (*e.g.*, a web site [17] or a slide show [43]), or to a particular development stage (*e.g.*, for web page redesign [10]).

3 THE USYBUS COMMUNICATION FRAMEWORK

To address the aforementioned shortcomings and the three quality factors (*i.e.*, compatibility, maintainability, and portability), we present UsyBus, a communication framework for supporting eye-tracking in interactive applications with parameterizable multiple inputs/outputs. For this purpose, UsyBus implements a data processing system designed on a multi-agent architecture [44], where intelligent agents [45] communicate with each other via a data exchange bus.

[1]See https://www.blickshift.com/

[2]See https://www.eyetracking.com/eyeworks-software/

3.1 Introductory Example

We hereby introduce a simple interactive application that serves as an introductory example for the remainder of this paper. This case study concerns an application dedicated to the registration of participants in an event. This application is intended to test the acceptability of the dynamic adaptation feature to end-users [31]. The user interface of the application is made up of five fields (first name, last name, street, zip code and city) for each participant to be registered (top right of Fig. 1). The application is adaptive, that is to say, it could dynamically swap order of some user interface fields (first name and last name, zip code and city) and groups of fields (personal details and contact details) depending on users' perceptions and actions. The adaptation engine applies a Perception-Decision-Action algorithm (PDA) [11, 33] to differentiate fields that have been watched at but left unused, fields that have been watched at and used, and fields that have been ignored. At run-time, the adaptation engine suggests to the application a new user interface layout, minimizing the time required to complete the actual user interaction path.

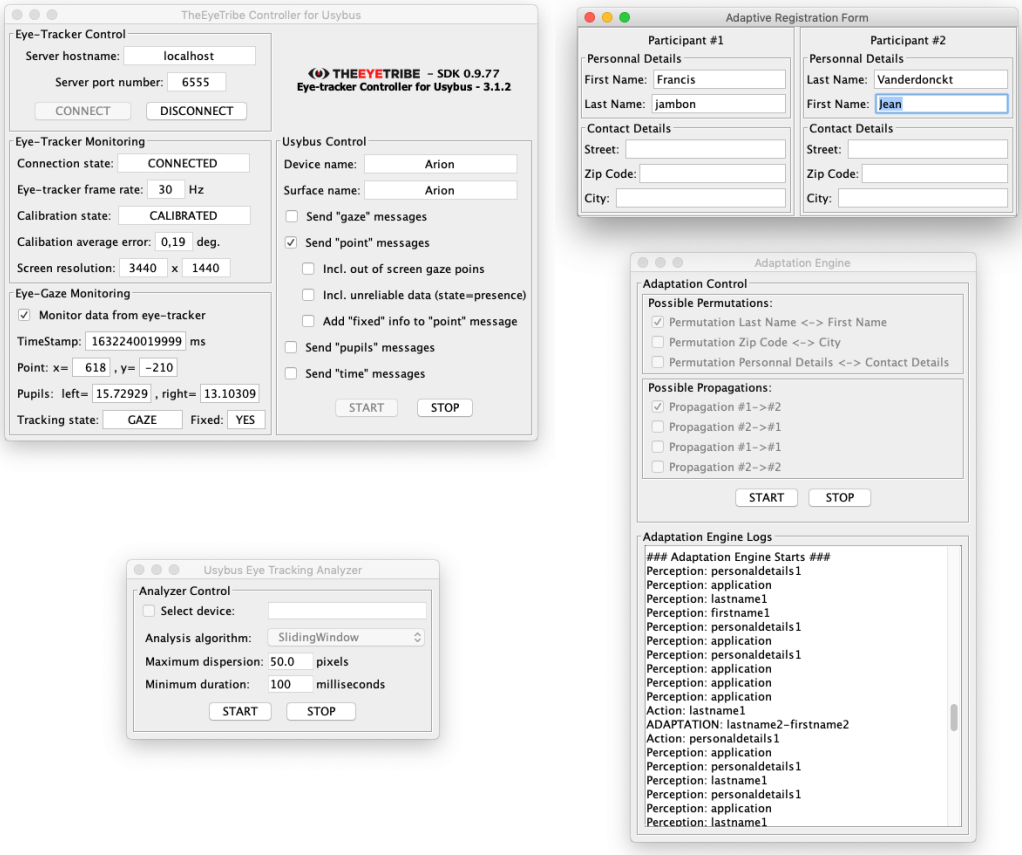

Fig. 1. Simple interactive application user interface.

For this purpose, the typical software architecture described in section 4.8 is instantiated to obtain the following specific interactive application architecture (Fig. 2):

- An Eye Tribe ET1000 eye-tracker, connected via its *Eye-Tracker Controller* (top left of Fig. 1), analyzes the user's gazes and sends the points –where the user is looking at– to the data exchange bus in real time (controllers are described in section 4.1).

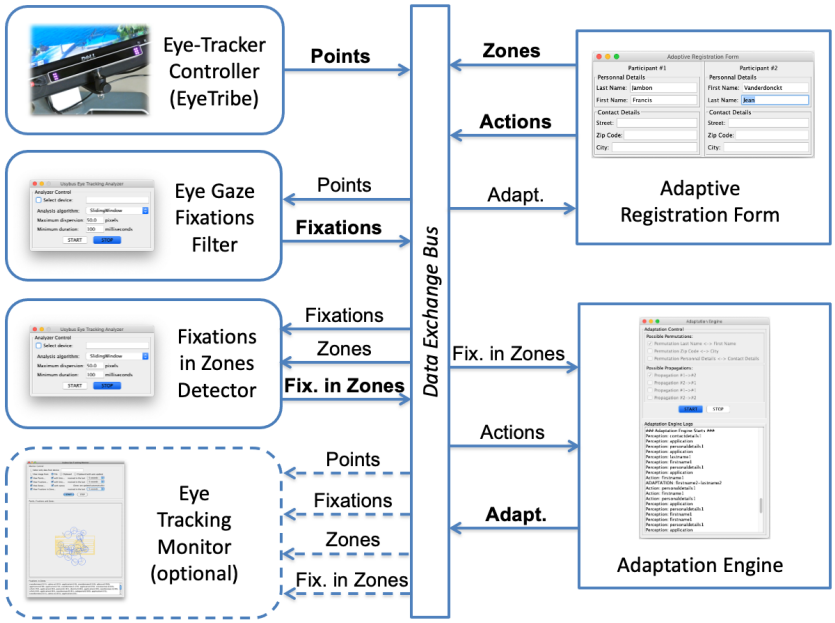

Fig. 2. Simple interactive application multi-agent architecture based on UsyBus.

- The *Eye Gaze Fixations Filter* (bottom left of Fig. 1) computes the fixations from the points (analyzers are described in section 4.3).
- The *Adaptive Registration Form* (top right of Fig. 1) sends the positions of the user interface fields and the actions of the user on these fields.

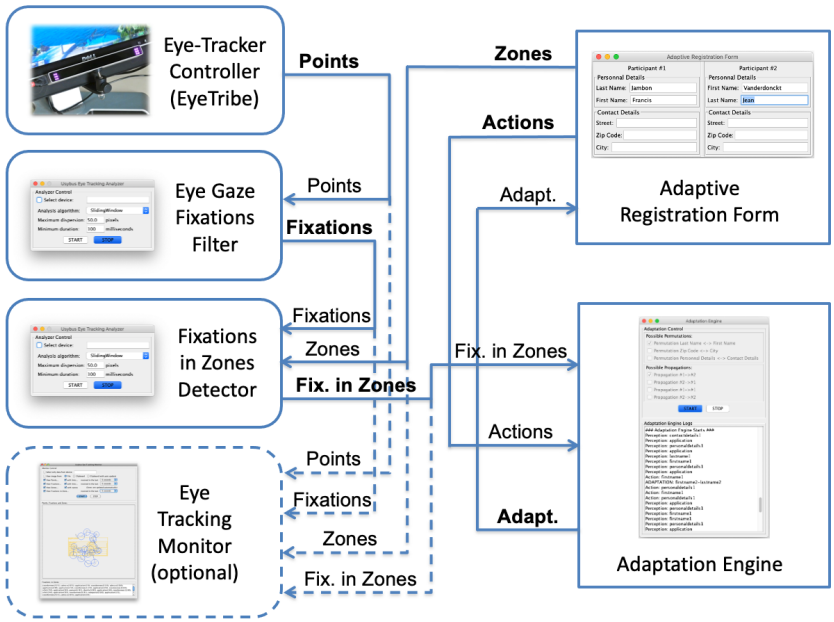

Fig. 3. Simple interactive application implicit dataflow between UsyBus agents.

- The *Fixation in Zone Detector* (bottom left of Fig. 1) uses the fixations (from *Eye Gaze Fixations Filter*) and the zones (from *Adaptive Registration Form*) to compute the fixations in zones (analyzers are described in section 4.3). Note that the *Eye Gaze Fixations Filter* and the *Fixation in Zone Detector* agents share the same user interface.
- The *Adaptation Engine* (bottom right of Fig. 1) uses the fixations in zones (from *Fixation in Zone Detector*) and the actions (from *Adaptive Registration Form*) to determine whether adaptation is necessary. If necessary, the agent sends the adaptation request to the data bus.
- The *Adaptive Registration Form* (top right of Fig. 1) uses the adaptation requests to change its user interface layout.
- In option, an *Eye Tracking Monitor* (not in Fig. 1) can display in real time the points, the fixations, the zones, and the zones viewed by the user (monitors are described in section 4.5).

This architecture defines de facto the dataflow (Fig. 3) between agents. As stated hereafter, this dataflow is implicit since the developer does not need to make explicit links between agents, but simply emit specified UsyBus data types or bind to these specified types.

3.2 UsyBus Multi-Agent Architecture

As the foregoing illustrative example suggests, the UsyBus model is a multi-agent software architecture [44] where intelligent agents interact with each other in an autonomous way. Moreover, UsyBus adopts a multiple-input multiple-output (MIMO) paradigm [5] (Fig. 4): any UsyBus agent can send data to the data exchange bus via one or many channels, and receive data in the same way. Channels are defined by UsyBus data types. Agents sending data to the bus are represented by arrows towards the data bus, while receiving agents (*e.g.*, for listening and collecting data) are represented by incoming arrows toward the agent. Agents connected to UsyBus have well-defined roles, they can emit and retrieve data on the data exchange bus without taking care about other agents, thus remaining autonomous. Sending agents do not need to know to which agent data is sent, and receiving agents do not need to know from which agent the data is coming from. For example, one sending agent retrieves raw oculometric data from an eye-tracker and sends formatted gaze point data (controller agents are described in section 4.1), another sending/receiving agent receives these points, calculates the fixations, and sends these fixations (analyzers agents are described in section 4.3), and at last, another receiving agent retrieves all these data and displays them (monitors are described in section 4.5).

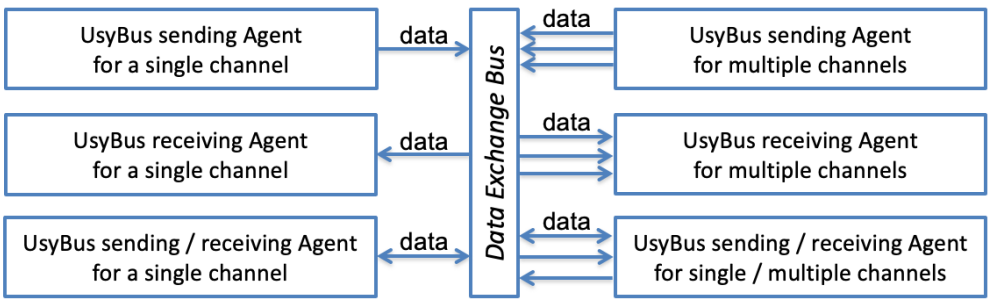

Fig. 4. Types of UsyBus agents in the multiple-input multiple-output (MIMO) paradigm.

3.3 Communications between UsyBus Agents

The UsyBus multi-agent architecture uses agent-to-agent communications based on UsyBus data types binding. Once receiving and sending agents are connected to the bus, the receiving agent

must bind to each type of data to be received from the bus. After that, the agent enters in a listening loop for any incoming message from the desired data types. At any time, the agent can unbind data types, and as a consequence, stops receiving associated messages. When all operations are over, the agent disconnects from the bus. As agents are autonomous, they react and decide independently of the actions of other agents.

A typical UsyBus agent that **receives** data should be implemented as follows (see Algorithm 1):

Algorithm 1: Receive Messages

```
create a databus object;
start databus connection;
bind to one or more data types;
while exist messages to receive do
    wait for message;
    process the message;
end
end the databus connection;
```

A sending agent sends data on the bus without worrying whether other agents have a binding to the data type or even connected. Conversely, receiving agents can bind or unbind data types whether or not a sending agent emitting the data type is connected. Since each sending agent is identified by its device name, each receiving agent can filter data only from any specific device. This mechanism supports multiple agents connected to the bus and using the same data types, but running for different applications or experiments.

A typical UsyBus agent that **sends** data should be implemented as follows (see Algorithm 2):

Algorithm 2: Send Messages

```
create a databus object;
start databus connection;
while exist messages to send do
    create a message (and specify its data type);
    send the message;
end
end the databus connection;
```

As the FIPA[3] organisation promotes the interoperability between heterogeneous software agents and agent-based systems, we also express UsyBus agents interactions according to this protocol. More especially, UsyBus agents interactions are specified using the AUML modelling language based on UML and compatible with the FIPA standard [22]. Fig. 5 illustrates the sequence diagram applying AUML between agents through the bus, and these are equivalent to Algorithms 1 and 2 defined previously.

[3]FIPA is a standard organisation agreed and recognised by the Institute of Electrical and Electronics Engineers (https://www.ieee.org) for agents and multi-agent systems.

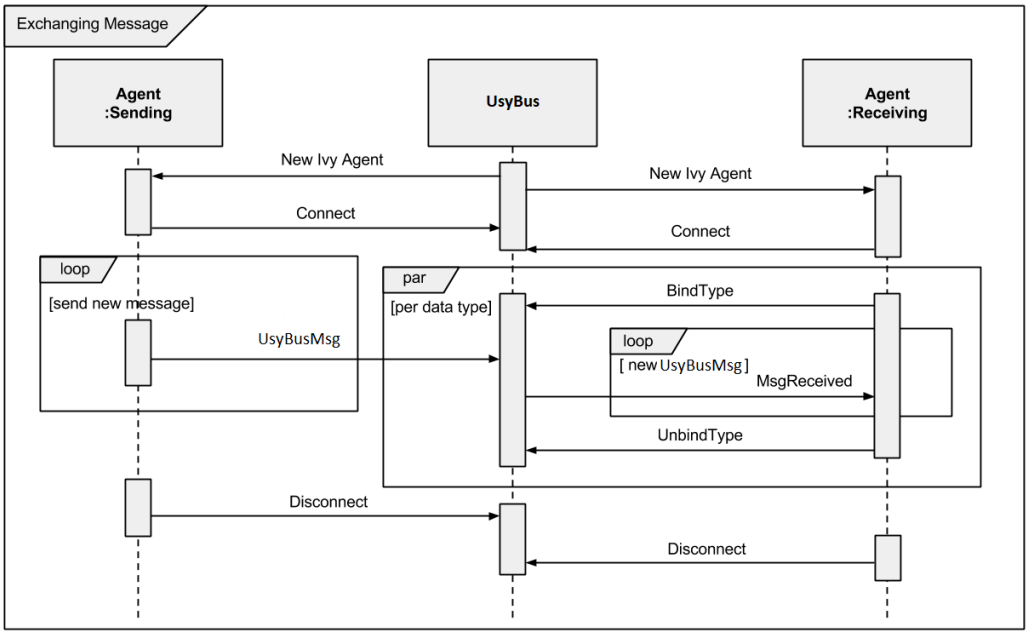

Fig. 5. UsyBus Agents Exchanging Message Interaction Sequence Diagram.

3.4 UsyBus Datagram and UsyBus Data Types

Data transmitted in messages that are exchanged between UsyBus agents must respect the UsyBus datagram format. The datagram is structured into two parts: the header that contains metadata, such as the version of the bus, the type of data and the origin of the data, and the payload that contains the data to be processed by the receiving agent(s).

Datagram structure uses a JSON-inspired format:
`UB2;type=data-type(:data-subtype(s))*;from=sender-application(;variable=value)+`

All fields are mandatory and must be completed in the order indicated:

- `UB2`: indicates the version of UsyBus datagram, here it is the version 2.
- `type`: defines the type of data, possibly with one or more sub-types indicated after and separated (by convention) from the type by the colon character ":" which is a reserved character (by convention) for types and sub-types.
- `from`: defines the name of the application that produces the data, it is usually the name of the application that implements the agent.
- The following fields define the data. The data format is of the form `variable=value`. If several data are transmitted, the separator character between these data is the semicolon character ";" which is a reserved character in the UsyBus datagram.
- There is no terminator character in the message.

To avoid compatibility issues with charsets, UsyBus datagram uses only US-ASCII characters. The semicolon character ";" is a reserved character and should never be used outside of its use as a separator.

The POSIX regular expression used to recognize a syntactically valid UsyBus datagram is:
`UB2;type=[^;]+;from=[^;]+(;[^;]+=[^;]+)+`

If the datagram format is the corner stone of the UsyBus framework, the data types must be considered as keystones. As developers can create UsyBus agents with no knowledge about the agents to which they will be connected, the definition of UsyBus data types must be precise and unambiguous both syntactically and semantically. Incorrect or incoherent definitions of data types may produce communications mismatches between agents in the dataflow, and as a consequence, unexpected behaviors of applications.

This is why a significant effort must be devoted to the specification and the documentation of data types, as for example in the Appendix A. For instance, the `eyetracking:Point` data type, hereafter described, defines the eye-gaze point relatively to the device for a defined timecode. It consists of the gaze coordinates in pixels on the screen where the user's eyes are looking at. This data type also has optional information specifying whether the gaze point is fixed.

```
type=eyetracking:point
Long tc                  – timecode (milliseconds since the epoch)
String device            – device name
Integer x                – point position on X axis (pixels)
Integer y                – point position on Y axis (pixels)
Boolean fixed (optional) – fixation information
```

3.5 Messaging System

Currently, UsyBus agents use the open-source Ivy[4] software library [12] as messaging library. The Ivy's authors describe it as "a simple protocol and a set of open-source (LGPL) libraries and programs that allows applications to broadcast information through text messages, with a subscription mechanism based on regular expressions". In other words, Ivy allows agents to exchange information as text. It is a (simulated) broadcast channel where agents send messages and other agents subscribe to the reception of messages. The subscription mechanism is based on binding –via regular expressions– to the contents of expected messages.

The implementation of UsyBus uses the binding mechanism of Ivy, limiting it to the header part of messages defining their type. Any UsyBus agent could be implemented directly with the Ivy library while respecting the UsyBus framework. However, to support the UsyBus use with Ivy, specific UsyBus libraries are implemented in Java and C#. Any other language can implement UsyBus, providing the Ivy library is available for that language (*e.g.*, Python, C++). In the future, agents could be implemented with other messaging libraries, such as ZeroMQ[5].

UsyBus libraries implement simple functions for bus connection, bus disconnection, type binding, type unbinding, message creation, data insertion into a message and data extraction from a message. These libraries also provide an abstraction of the messaging system used. As a consequence, UsyBus libraries allow the implementation of UsyBus agents in a easier, faster, less error-prone ways and improve independence from the messaging system used. As an illustration, an implementation of a simple application that use the Java UsyBus library is detailed in Appendix B.

4 REUSABLE USYBUS AGENTS

A substantive set of UsyBus agents has been developed within research and development projects. Agents can be specific to a project or reusable across projects. On a functional point of view, agents

[4]See https://www.eei.cena.fr/products/ivy/
[5]See https://zeromq.org

can implement interfaces with devices as eye-trackers or third-party applications as simulators. In these cases, most of these agents are generally limited to providing data to the data exchange bus. Moreover, these agents, mainly these that interface with eye-trackers, are reusable. Agents can also implement monitoring or data collection functions. In these cases, they are limited to retrieving data from the data exchange bus. These agents, in their great majority, are reusable. Another important function is analysis. These agents usually retrieve data from the data exchange bus and could, in return, display analysis results and/or provide these results back on the data exchange bus. These agents are reusable, as long as the analysis they provide is used in another project. At last, agents may be domain specific. These agents are developed for one project, thus limiting their reusability. This section proposes a representative overview of existing agents with a focus on reusability.

4.1 Eye-Tracker Controllers Agents

Eye-tracker controllers agents act as bridges between the eye-tracking device and the data exchange bus, establishing independent communication with respect to any eye-tracker. The Tobii Eye-Tracker Controller (Fig. 6) is a controller agent for Tobii screen-based eye-trackers. Developed in C#, its implementation is based on the Tobii SDK 2.0[6] library. Therefore, that agent is compatible with all Tobii "50 Series" devices, as Tobii 1750 eye-trackers. In the same way, the EyeTribe Controller (Fig. 7) interfaces any "Eye Tribe" eye-tracker to the data exchange bus. The controller was developed in Java with the library provided by Eye Tribe[7]. Similarly to the previous controller agents, the EyeWorks Controller (Fig. 8) is specifically tailored for EyeWorks compatible eye-trackers, such as the Seeing Machines Fovio[8] eye-tracker. It is implemented in C# with the EyeWorks[9] *AppConnect* SDK library.

For all eye-tracker controllers described above, the controllers functions include device connection/disconnection and status retrieval such as connection status, calibration status, frame rate and average calibration error. Eye-tracker controllers also have functions to choose which data should be sent to the bus. The main data that is sent to the bus is the *Point* message, that refers to the position of the gaze on the screen. Other messages are the *Pupils* message that sends the left and right pupils diameters, and the *Device* message that gives the resolution of the screen used by the eye-tracker.

As synchronisation of gaze data with other data as zones is critical, a *Time* message that sends internal eye-tracker clock time is also provided. As the clock of an eye-tracker usually relies on the clock of its image sensor, the drift depends on the quality of that clock and on the physical environment of the device, thus producing different rates between eye-tracker and computer clocks. Drift may be an issue if the computer clock is used for other messages, as *Zone* message, because a good synchronisation is requested to calculate fixations (reference = eye-tracker clock) in zones (reference = computer clock).

That is why controller agents display a clock synchronisation status which indicates the drift between the eye-tracker clock and the computer clock used as a reference. Warnings display that status in three colors (Fig. 6 and 8): if the drift is smaller or equal to 100 milliseconds, it is considered negligible (*Green*); if the drift is higher than 100 milliseconds, but smaller or equal to 1000 milliseconds, it is considered acceptable (*Orange*); if the drift is higher than 1000 milliseconds, it is considered as too high, thus no longer acceptable (*Red*). The EyeTribe SDK uses the host computer system clock for gaze data, and therefore its controller does not need such a status indicator.

[6]See https://www.tobiipro.com
[7]See https://theeyetribe.com
[8]See https://www.design-industry.com.au/seeing-machines/
[9]See https://www.eyetracking.com/eyeworks-software/

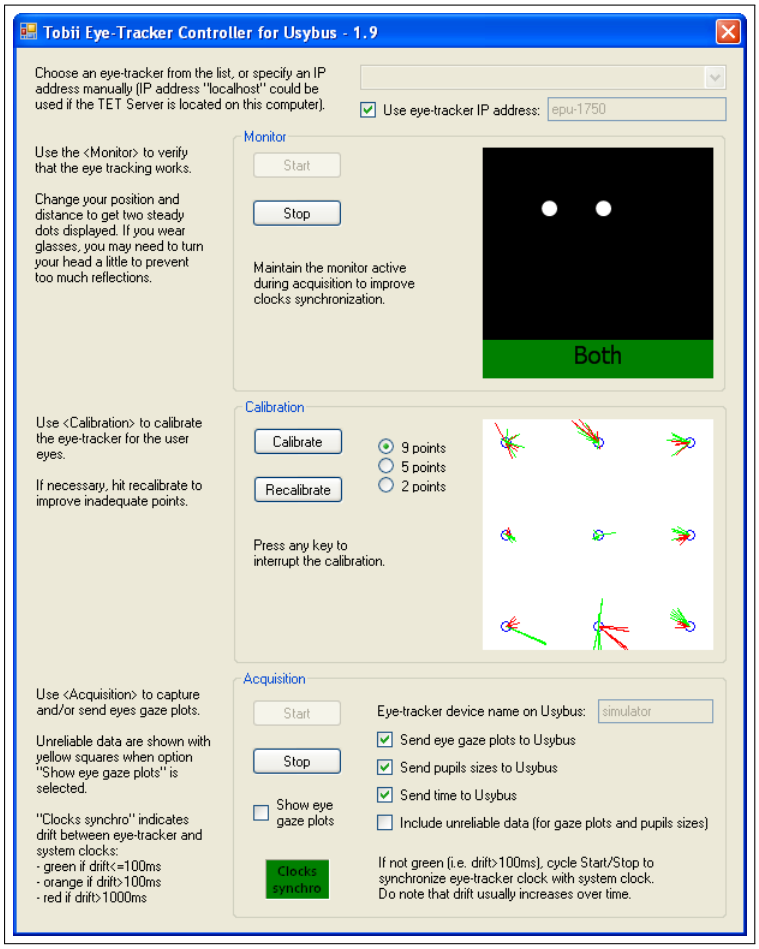

Fig. 6. Tobii 1750 Controller user interface.

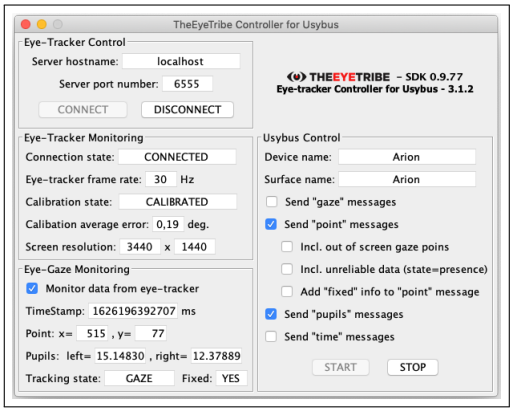

Fig. 7. EyeTribe Controller user interface.

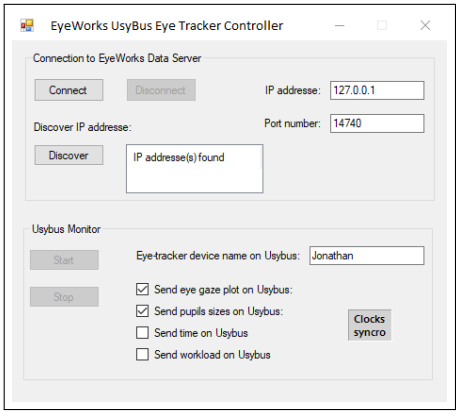

Fig. 8. EyeWorks Controller user interface.

Controller agents can also implement specific functions provided by eye-trackers SDK. For instance, the Tobii Controller (Fig. 6) features a “Calibration” option to calibrate the eye-tracker for

the user's eyes geometry, without having to use another application. The EyeTribe Controller (Fig. 7) provides a "Tracking state" option to verify whether the eye-tracker works properly. The EyeWorks Controller (Fig. 8) can send an Index of Cognitive Activity (ICA) in *Load* messages. Compatible only with the Seeing Machines Fovio eye-tracker, the ICA is calculated by the Cognitive Workload Module[10] of the EyeWorks software, and is based on variations of pupils diameters [35].

Another agent, the Eye-Tracker Mouse Simulator, can be used to simulate eye movements with mouse movements. This agent samples the movements of the mouse cursor on the screen, at the frequency of the simulated eye-tracker, and sends Point messages on the data exchange bus. This agent is typically used in the early stages of the development process, to test applications without needing to use an eye-tracker. Thanks to the versatility of the UsyBus framework, the substitution between an eye-tracker controller and the mouse simulator can be done without code modification.

Today three eye-tracker controllers are implemented, others can be implemented on demand. The implementation of a new controller in straightforward: the agent must use the eye-tracker manufacturer's SDK to provide on UsyBus at least the *Point* message. As an example, a new version of the Tobii controller, compatible with the new Tobii Pro SDK (*e.g.*, for Tobii Pro X2, X3, Spectrum, Nano and Fusion) is under development in Python 3.

The development of controllers for wearable eye-trackers (glasses) must take into account a specific feature of these devices. While screen-based eye-trackers returns gaze points in the screen coordinates, the wearable eye-trackers usually returns gaze points in the scene camera coordinates. As the scene camera is usually attached to the glasses frame, to be compatible with the *Point* message, controllers for wearable eye-trackers must transform gaze coordinates from scene camera coordinates to screen (or real world) coordinates in real time. This feature, called online real-world mapping, is available in the SDK of some eye-trackers (*e.g.*, Pupil Core[11]). Alternatively, one solution might be to implement the coordinate transformation function [2] in a specific UsyBus agent.

4.2 Widget Tracker Library/Agent

The Widget Tracker library is a utility library that tracks the positions of the widgets in a graphical user interface, and sends the geometry of these widgets as *Zone* messages on the bus. The library is not, strictly speaking, an agent since is it not a stand-alone application, but the library, when used by an application, enables that application to automatically create an agent on the bus. The library is compatible with any application that uses the Java Swing[12] graphical library.

From an implementation point of view, the Widget Tracker should be linked with the main JFrame component of the user interface. It traverses the Java Swing user interface components tree from root (*e.g.*, JFrame) to leaves (*e.g.*, JButton, JLabel). All named widgets, *i.e.*, widgets with neither null nor empty *name* attribute, that are included in that JFrame component will automatically be sent as *Zone* messages on the bus. In this way, the name attribute of Swing widgets can be used to implement a simple filtering mechanism, in order to allow programmers to select only the widgets of interest in the user interface. Of course, any modification of the main frame or widgets (*e.g.*, creation, destruction, change is shape or position) is detected via the Java event listeners, and consequently, *Zone* update messages are sent to the bus if necessary.

The implementation of a Widget Tracker library for another graphical library is possible, if only the library provides ways to enumerate the set of visible widgets, has introspection functions to determine the widgets positions relative to the screen coordinates, and has ways to be notified of changes in the user interface layout.

[10] See https://www.eyetracking.com/cognitive-workload/
[11] See https://pupil-labs.com
[12] See https://docs.oracle.com/javase/8/docs/api/javax/swing/

4.3 Eye Tracking Analyzer Agents

The Eye Tracking Analyzer is composed of two agents, an Eye Gaze Fixations Filter and a Fixations in Zones Detector, which share the same user interface (Fig. 9). The Eye Gaze Fixations Filter agent captures data concerning eye gaze points and returns new data concerning the eye fixations to the bus. The analysis can be performed with several algorithms such as:

(1) The original *Widdel* algorithm published as the Dispersion-Threshold Identification (I-DT) [37] algorithm, one of the most frequently used algorithm to identify fixations as groups of consecutive points within a particular dispersion.
(2) The *Sliding Window* algorithm, an experimental algorithm based on I-DT but with a sliding window to improve the inclusion or rejection of the first and last gaze points of a fixation;
(3) The *Minimum Fixation* algorithm, another experimental algorithm based on a search for fixations whose dispersion is of minimum size.
(4) The *Overlapping Fixations* algorithm, which detects the starting of fixations, called "premises of fixations" [34]. Contrarily to other detection methods which detect fixations only when they are completed, the overlapping fixations algorithm uses a method called "on-line fixation" that can detect fixations in real time as soon as they are potentially detectable [30].

The *Fixations in Zones Detector* agent captures the fixations and zones, and returns the fixations in zones, that is which zones have been fixed, and for how long time the fixation lasts in them.

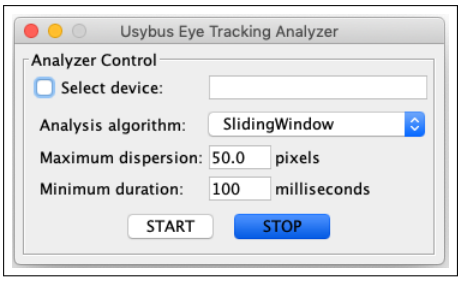

Fig. 9. Eye Tracking Analyzer user interface.

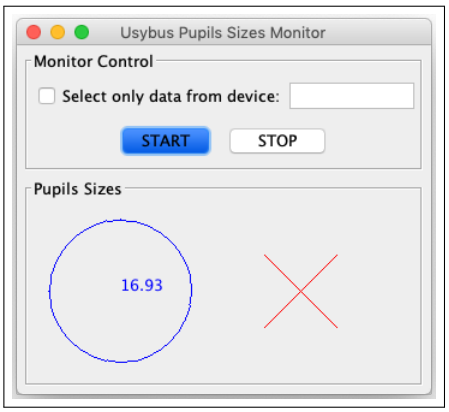

Fig. 10. Pupils Sizes Monitor user interface.

4.4 Pupils Sizes Monitor Agent

The Pupils Sizes Monitor agent is a simple monitoring agent displaying the pupil size for both eyes. The interface reproduced in Fig. 10 shows a blue circle representing the pupil size and its diameter value (in device units), or a red cross when no data is available on the bus, when the data might be of too low quality, or when the user blinks. In this way, the size of the pupils can be observed in real time. This agent is often used to monitor whether the eye-tracker "sees" the user, since pupil size is the first data an eye-tracker captures when detecting user's eyes.

4.5 Eye Tracking Monitor Agent

The Eye Tracking Monitor agent is a monitoring agent that displays in real time the gaze points, the fixations, the zones, and the fixations in the zones (Fig. 11). As the Eye Tracking Monitor agent displays almost all eye-tracking related data that are available on the bus, it is primarily used for debugging or for monitoring an experiment. The monitor can show/hide the points, the fixations, the zones and/or the fixations in zones on demand. Other options are available as drawing or not

the links between gaze or fixation center points, displaying data for a certain amount of time, and displaying a background image (the contents of the screen associated with the eye-tracker) from a file or from the clipboard. When used with the clipboard, the background image features an auto-update function that automatically updates it if a new image is set in the clipboard (*e.g.*, when print-screen key is pressed). This function can also be used to automatically update the monitor background image in real time, if an application triggers –by software– print-screen to the clipboard each time it changes the layout of its user interface.

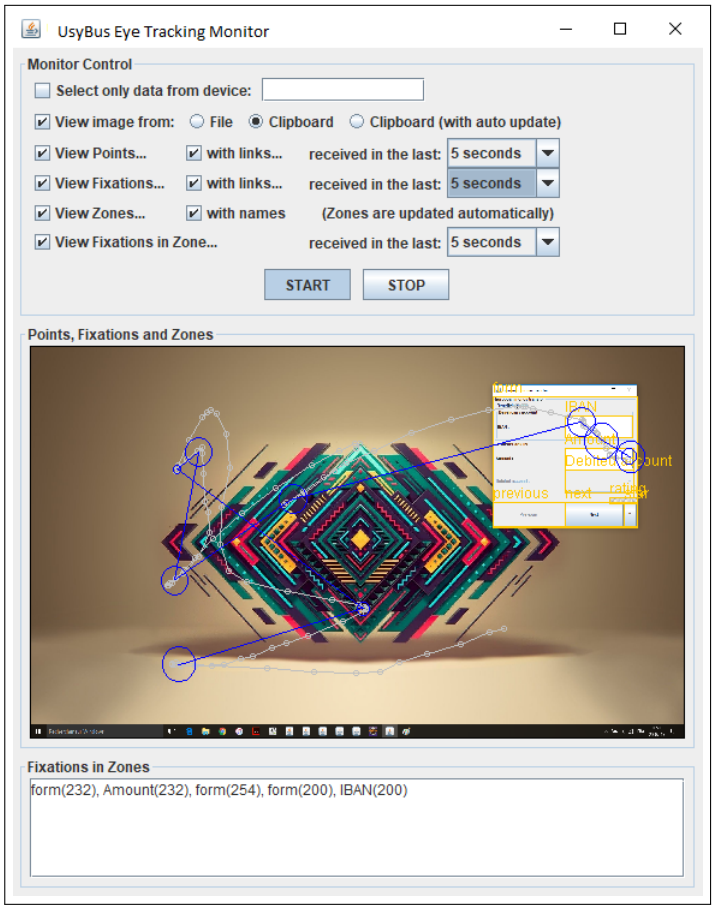

Fig. 11. Eye tracking Monitor user interface.

An expanded version of this agent, the Expanded Eye Tracking Monitor was developed for improving the feedback analysis of a session (Fig. 12). First, a screen shot of the monitor window can be captured at any time either manually or automatically in low or high resolution in different formats (*e.g.*, PNG image file), with or without data displayed on the monitor at that time: the background image, the gaze points, the fixations, and the zones. Selecting the "Auto print" function triggers a screen-shot capture each time the monitor displays a new image. Second, a filter can select and display only zones that contain Java Swing widgets (Fig. 13). This automatic assignment of a zone to any such widget is a facility particularly suited for user interface analysis as it eliminates the need to manually define all these zones, a tedious task when many widgets are included.

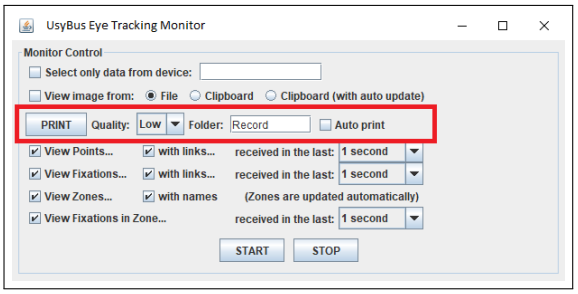

Fig. 12. Expanded Eye Tracking Monitor user interface.

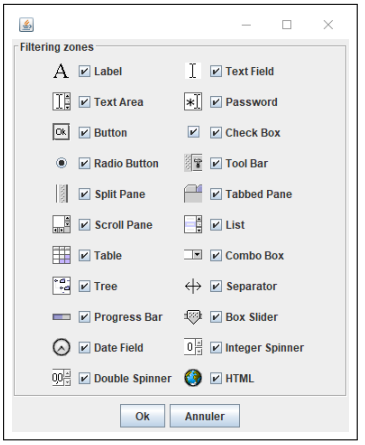

Fig. 13. Zone Filter user interface.

4.6 Cognitive Load Monitor Agent

The Cognitive Load Monitor agent is a monitoring agent that observes in real time the evolution of the left and right Index of Cognitive Activity (ICA) and renders them via a line chart (Fig. 14). For example, in the case of a training simulator, an instructor could analyse, during a simulation, whether a trainee has a higher cognitive activity in comparison with any cognitive activity reference, like the one of an experienced user.

The Cognitive Load Monitor agent retrieves *Load* messages related to the cognitive activity from the bus. The ICA line chart displays ICA as a function of time as follows: the y-axis of the ICA measurement varies between 0 (the lowest activity) and 1 (the highest activity measurement) and the x-axis expresses the experiment over time in seconds since the first data received. A "Reset Time" function sets the current time to a new zero time reference point, reflected by a vertical red line on the graph. Furthermore, the mean between left and right ICA can be computed, adjusted, and visualized. The scale of the graph can be changed by increasing or decreasing the number of points on the graph.

In addition, the agent allows defining tasks. Defining such tasks turns out to be an important prerequisite for the cognitive activity analysis since tasks could have different realisation solutions and/or different levels of difficulty. Starting a new task can be achieved manually via the "New task" function. New tasks are set for the current time and are depicted by a purple vertical line on the graph. The task name is written on the top of this line.

With that monitor, an observer knows exactly when tasks are started, and which task correspond to each part of the ICA line chart. A time schedule can be specified (Fig. 15) via the "Task Scheduling" function. A table with the tasks names and their starting times in seconds, relative to the zero reference time, can be filled in to organise the tasks scheduling by adding, by removing, by modifying, and by moving the scheduling lines. Any completed schedule can be saved in a XML file to be loaded for future usage. Once loaded, the XML file is parsed and *Task* messages are sent on the bus to communicate this information to other agents.

For this UsyBus agent, both ICA and Tasks visualizations are mixed in an unique monitor. As a consequence, the Cognitive Load Monitor is not as generic as it could be. This choice was motivated to facilitate the visualisation of both ICA and Tasks for a specific experiment. However, to maintain message-level genericity, the *Load* and *Task* messages have been defined as different, so more generic monitoring agents could be implemented in the future.

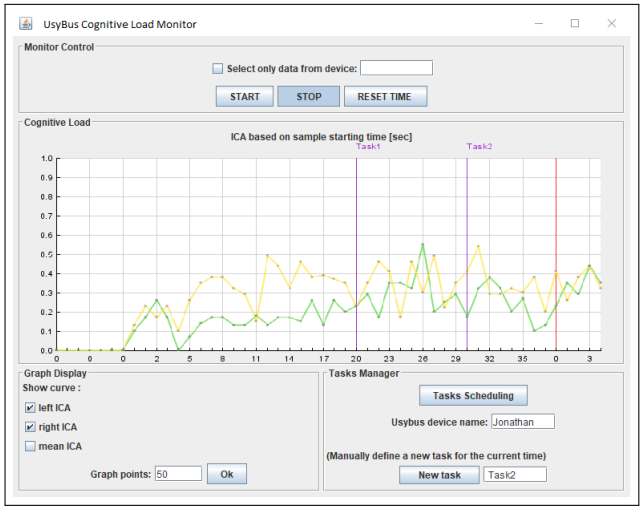

Fig. 14. Cognitive Load Monitor user interface.

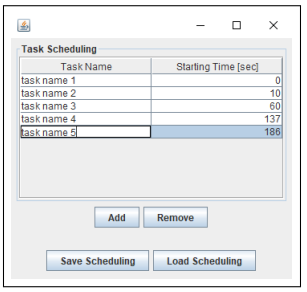

Fig. 15. Task Scheduling user interface.

4.7 Message Recorder and Message Player Agents

The Message Recorder (Fig. 16) and Message Player (Fig. 17) are utility agents. They respectively records data from the data exchange bus, and replays (sends) them on it. In this way, any experiment can be stored and replayed step by step or through filters, for example to analyze the data in more detail after the experiment. All messages are saved in XML format and can therefore be analyzed by third-party software. The list of data to be recorded can be configured at any time by manual editing, by loading a file, or by a combination of the two. The recording agent is also capable of retrieving all types of data passing through the bus. As for other listening agents, the recorder can filter data coming only from a selected device. The player agent can read any XML-compliant file created by the recorder agent, and can replay the messages on the bus. At replay, original or updated (to the current time) timecodes can be used. The repeat option allows to repeat the sequences when ended, and is usually used for debugging.

4.8 Agents Usage and Typical Architecture

Based on the UsyBus framework defined in section 3, the features of the reusable agents detailed in this section suggest de facto a typical usage scenario and architecture for these agents. This typical usage scenario could be as follows:

- An Eye-Tracker Controller agent sends at least *Point* messages on the bus.
- A domain specific application sends *Zone* messages on the bus, for instance using the *Widget Tracker* library. This application can also send some domain-specific data.
- An Eye Gaze Fixations Filter agent uses *Point* messages to calculate *Fixation* messages.
- A Fixations in Zones Detector agent uses *Fixation* and *Zone* messages to determine *Fix. in Zone* messages.
- *Fix. in Zone* and domain specific messages are used by a domain specific analyzer.
- In option, eye-tracking data can be monitored by an Eye-Tracking Monitor agent.

This usage scenario defines a typical architecture (Fig. 18) where rectangles with rounded corners indicate reusable agents, classical rectangles indicate specific agents, dotted rectangles indicate optional agents, and data types are shown in bold when generated.

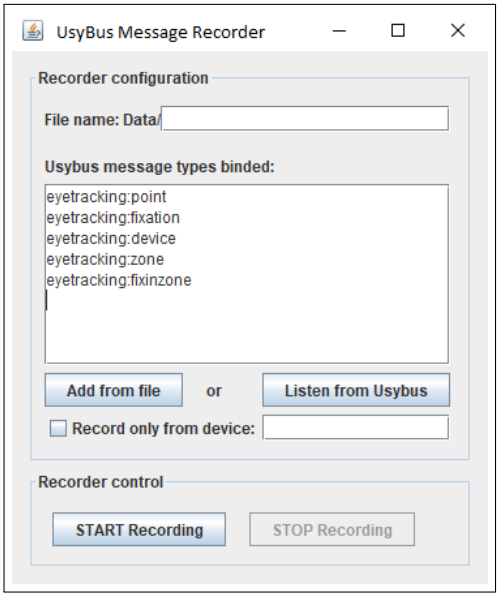

Fig. 16. Message Recorder user interface.

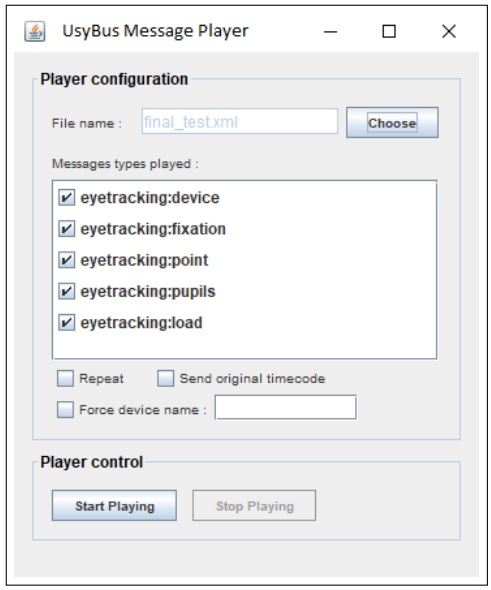

Fig. 17. Message Player user interface.

It is important to notice that the UsyBus framework does not explicitly define a data-flow, but the data bindings that are performed (even at run-time) by agents does define a data-flow implicitly. This characteristic highlights a major advantage of the UsyBus framework, *i.e.* it is possible to design an agent by focusing only on the types of data it requires and the types of data it must provide, without needing to know the whole system architecture nor data-flow, and in particular the other agents providing or using these types of data. In other words, the architecture and the corresponding data-flow are based only on the definition of the types of data exchanged. As an illustration, table 1 summarizes the UsyBus data types manipulated by core reusable agents (see also appendix A for more details). From this typical architecture, two case studies of existing applications are detailed in section 5 to illustrate the actual use of UsyBus agents.

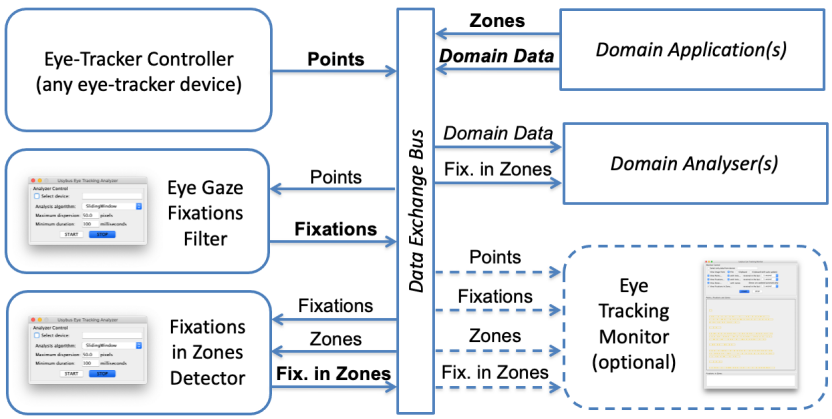

Fig. 18. Typical multi-agent architecture based on UsyBus.

Agent \ Message Type	Point	Pupils	Fixation	Zone	Fix. in Zone	Load
Eye-Tracker Controller	E	(E)				(E)
Widget Tracker library				E		
Eye Gaze Fixations Filter	R		E			
Fixations in Zones Detector			R	R	E	
Pupils Sizes Monitor		R				
Eye Tracking Monitor	R		R	R	R	
Cognitive Load Monitor						R
Message Recorder/Player	(R)/(E)	(R)/(E)	(R)/(E)	(R)/(E)	(R)/(E)	(R)/(E)

Table 1. Overview of main data types used by UsyBus agents for eye-tracking: "R" denotes a retrieving agent, "E" denotes a emitting agent. Brackets indicate an optional data type.

5 CASE STUDIES

Over the years, about a dozen of interactive applications were developed with the UsyBus framework. These applications architectures instantiate the typical UsyBus architecture presented in section 4.8 and reuse several UsyBus agents detailed in section 4. To exemplify the UsyBus framework usage, this section details two illustrative cases studies: an eye-tracking interactive application for information search on the web, and the prototype of an intelligent tutoring system based on an instrumented flight simulator.

5.1 The SearchNGaze Case Study

SearchNGaze [40] is an eye-tracking interactive application [18] that was developed in the context of Information Retrieval. It is a proof of concept to study how on-line eye movement analysis could improve the performance of a web search engine.

With classic web search engines, after the query of a user, the system usually returns a list of results composed of extracts of documents –called snippets– of supposedly relevant documents. The user chooses the document that seems most interesting by clicking on the corresponding snippet, and then consults that document. If the document is not relevant, the user can return to the results page, and chooses another document, etc.

This process can be time-consuming if the most relevant results are not ranked among the top results. Moreover, the initial query may give few relevant results, for instance when the terms of the query are ambiguous. As an illustration, if a user types "earth" for a query, her/his intention could be "earth" as a planet or "earth" as ground. That is why, knowing the implicit interests of the user (*e.g.*, planetology), and adding specific terms to the initial query (*e.g.*, "earth" becomes "earth planet"), may significantly improve the results returned by a search engine. The main objective of the SearchNGaze application is to verify this hypothesis via user experiments.

The application (Fig. 19) uses eye movement analysis to detect words on the results page that seem relevant to the user implicit interests, and selects one of these words to reformulate the initial query, expecting more relevant results. The implementation of the SearchNGaze application raises two main technical challenges. First, the eye movement analysis must be performed on-line since the identification of pertinent words must be performed while the user is reading the results page. Second, the user interface –the results page layout– changes depending on the query, and as a consequence, the words and their positions in the page are non predictable and must be determined dynamically.

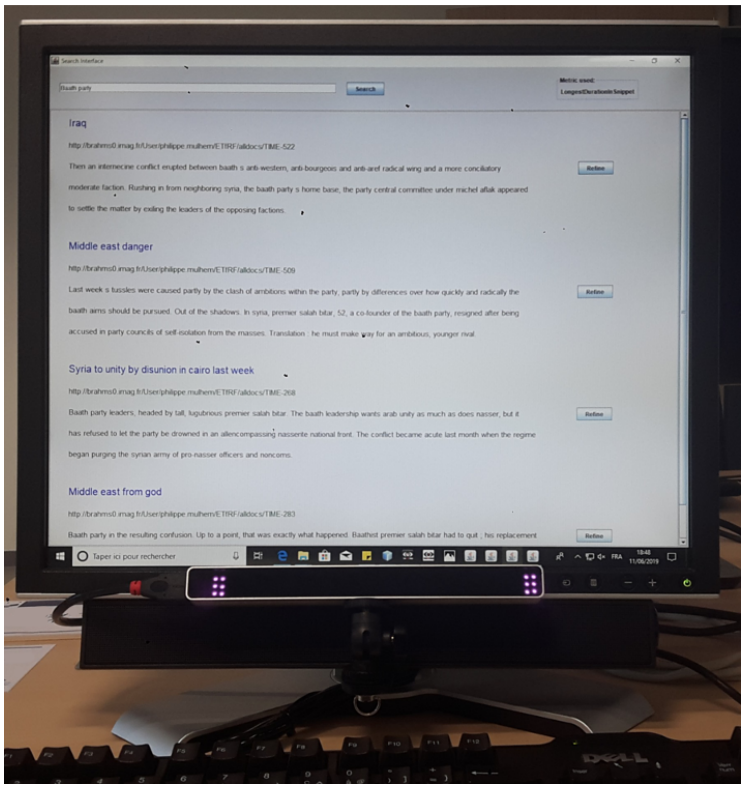

Fig. 19. SearchNGaze experimental setup.

Fig. 20 details the multi-agent architecture of the SearchNGaze application, with focus on UsyBus agents: rectangles with rounded corners indicate reusable agents; classical rectangles indicate specific agents; dotted rectangles indicate optional agents; data types are shown in bold when generated. Most eye-tracking functions, *i.e.*, eye-tracker control, gaze capture, fixation calculus and fixation in word detection are performed by UsyBus reusable agents: Eye-Tracker Controller, Eye

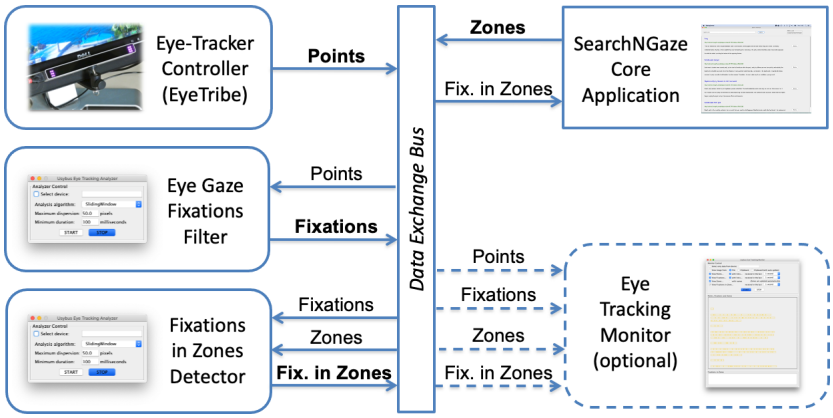

Fig. 20. SearchNGaze multi-agent architecture based on UsyBus.

Gaze Fixations Filter and Fixations in Zones Detector. As the user interface is developed with Java Swing, the Widget Tracker library is used to automatically send user interface zones to the bus. For this case study, the zones are the words of the snippets. Another reusable UsyBus agent, the Eye Tracking Monitor, can be used to monitor all eye-tracking related data during the user experiments.

With the help of these reusable UsyBus agents, the SearchNGaze core application role is limited to sending snippets words positions to the bus, and in return, retrieving the subset of fixated words. It is therefore possible, for the core application, to determine the most pertinent word from the fixated words set (via a specific indicator), to extend the initial query with that pertinent word, to send a refine query to the information search system, and thus to potentially improve the results list without explicit intervention by the user. Thanks to the UsyBus framework and the portfolio of reusable agents, SearchNGaze's development effort related to eye-tracking functions has been minimal, and the design of the application has been greatly facilitated.

As an illustration, Fig. 21 shows a screenshot of the Eye Tracking Monitor during an experiment, with in overprint, the eye-tracking data. Yellow rectangles enclose snippets words, and blue circles are centered on fixations. In this example, the documents and the query come from the TIME magazine test collection.

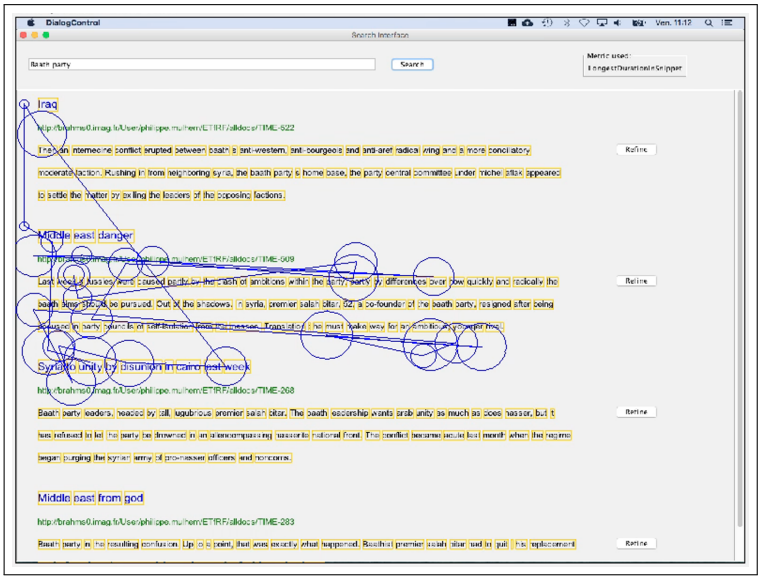

Fig. 21. Eye Tracking Monitor used for the SearchNGaze application.

5.2 The PILOTE2 Case Study

The PILOTE2 case study [32] is an instrumented flight simulator developed in the context of Technology Enhanced Learning Environments. This proof of concept aims to study the feasibility of pilots' skills on-line diagnostics, based on the analysis of their perceptive and gestural activities. The final objective of PILOTE2 is to develop an Intelligent Tutoring System based on this on-line skills diagnosis.

The PILOTE2 application (Fig. 22) was designed with a PC-based flight simulator (Microsoft Flight Simulator X) and uses consumer flight control commands (Saitek Pro Flight Cessna). The multi-agent architecture of the PILOTE2 application is details in Fig. 23 with focus on UsyBus agents: rectangles with rounded corners indicate reusable agents; classical rectangles indicate

Fig. 22. PILOTE2 instrumented simulator experimental setup.

specific agents; dotted rectangles indicate optional agents; data types are shown in bold when generated. A specific UsyBus agent (Flight Simulator Interface) was developed to interface with the flight simulator in order to capture pilot's actions on control commands (*e.g.*, flaps extension switch activation), simulator states (*e.g.*, airspeed of the aircraft), and zones on the dashboard (*e.g.*, position of the airspeed indicator on the screen). If the flight simulator is changed, only this agent needs to

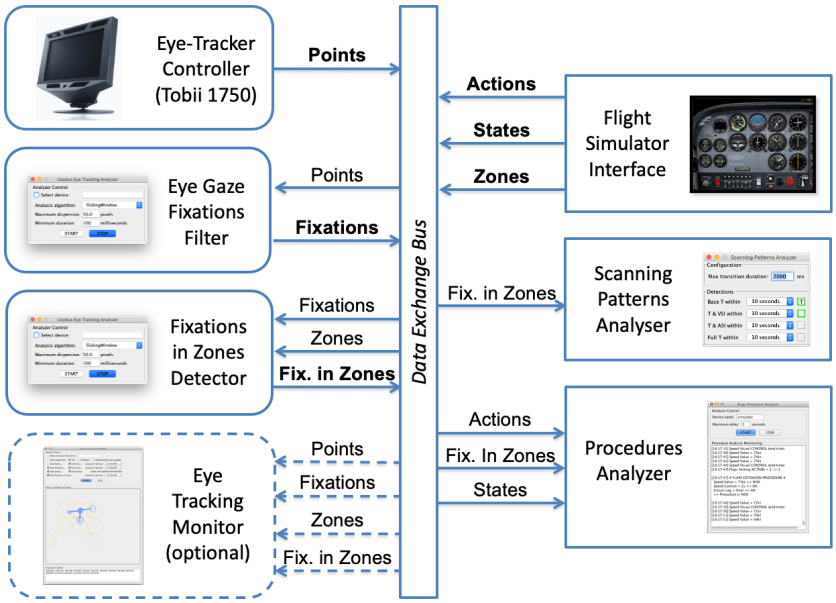

Fig. 23. PILOTE2 multi-agent architecture based on UsyBus.

be changed. Although more complex than the SearchNGaze case study, the PILOTE2 architecture reuses at least two eye-tracking related UsyBus agents: Gaze Fixation Filter and Fixations in Zones Detector. A new Eye-Tracker Controller agent was developed to interface a Tobii 1750 eye-tracker with the data exchange bus. The Eye Tribe eye-tracker used for SearchNGaze could have been used, but for better system integration, a Tobii 1750 was preferred. Another reusable UsyBus agent, the Eye Tracking Monitor, is used to monitor all eye-tracking related data.

The pilots' dashboard scanning patterns are analyzed by a domain-specific UsyBus agent (Scanning Patterns Analyzer) to detect whether the pilot used the recommended "T" shape pattern for reading dashboard instruments (Fig. 24). Another domain-specific UsyBus agent (Procedures Analyzer) was developed to analyze some representative piloting procedures (Fig. 25). For instance, the flaps extension procedure requires the pilot to first read the airspeed (data = fixation in a specific zone), if this airspeed is under a certain threshold (data = simulation state), he/she can press the flaps extension switch (data = action on a control command). To do this, the Procedures Analyzer agent combines fixations in zones, actions, and simulation states that are available on the data exchange bus. In addition, the pilots' gaze activity can be monitored in real time using the Eye Tracking Monitor agent (Fig. 26).

As in the SearchNGaze case study, the PILOTE2 development effort related to the eye-tracking functions has been minimal, and limited to the development of the new Tobii 1750 controller agent. As a result, the design of the application has been greatly facilitated thanks to the UsyBus framework. In addition, the Tobii 1750 controller agent has integrated the portfolio of reusable UsyBus agents.

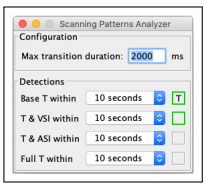

Fig. 24. PILOTE2 Scanning Patterns Analyzer.

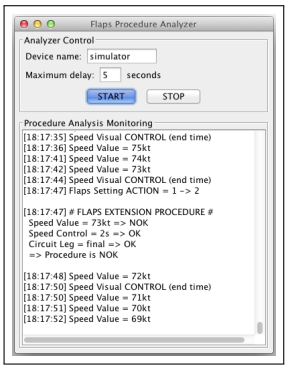

Fig. 25. PILOTE2 Flaps Procedure Analyzer.

Fig. 26. Eye-Tracking Monitor used for the PILOTE2 experiment.

6 CONCLUSION AND FUTURE WORK

This paper presents UsyBus, a framework for supporting eye-tracking studies in a wide variety of contexts of use, ranging from the evaluation of user interfaces to the development of gaze interaction applications. The framework consists of a series of autonomous intelligent agents, structured in a multi-agent software architecture. Agents are sending and/or receiving formatted messages from/to the data exchange bus, and are implicitly organized based on types they send/receive.

The portfolio of existing reusable UsyBus agents features a large set of eye-tracking related functions such as gaze capture, fixation filtering, zone extraction, fixation in zone detection, eye-tracking data monitoring, data recording... For instance, the Widget Tracker library/agent automatically identifies widgets of a graphical user interface as zones of interest, that are automatically updated if the user interface layout changes.

UsyBus aims at addressing particularly three important ISO 25010 software quality properties, *i.e.*, compatibility (*e.g.*, with different eye-trackers, applications, and environments), maintainability (*e.g.*, by parameterization of the agent input/output), and portability (*e.g.*, for replacing the agent responsible for a function by another one without compromising the overall stability of the architecture).

Although these three properties are explicitly addressed, the framework will always depend on the extent to which the various agents developed are available: the more agents are available, the better these properties will be satisfied. For instance, if a new eye-tracker arrives on the market that benefits from more advanced capabilities, there is still the need and the dependency of an agent able to extract raw data from the eye-tracker, and transform them into formatted UsyBus messages to be sent to the bus. However, this task needs to be done once.

The UsyBus framework is based on two key elements: the datagram format which defines the syntax of data exchanges between agents, and the data types which define both syntactically and semantically the payload content of the datagrams. Data types also participate to the definition of the implicit data flows between agents, via the data types binding mechanism. This latter element is also the keystone of the UsyBus framework, as the definition of these data types must be precise and unambiguous both syntactically and semantically, to prevent communications mismatches between agents in the data flow. As developers can create UsyBus agents with no knowledge about the agents to which they will be connected, the main data types must be defined and updated by centralized documentation.

Perspectives of this work are twofold. First, new developments will be engaged. Theses developments concern new controller agents for new oculometers, and in particular for mobile eye-trackers (*e.g.*, glasses), because this category of eye-trackers raises significant issues with real-world mapping of images from the scene camera. Developments also relate to better fixation filter algorithms from the literature. We are also concerned by solving time synchronisation issues that remain when different real-time clocks are used for gaze and zone. A option could be, for instance, to implement of the Network Time Protocol (NTP) in UsyBus agents.

A second perspective concerns the diffusion of the UsyBus framework. After gaining some maturity, we are starting the advertising of the framework and its diffusion. For this purpose, we are improving two aspects: a clear framework for synchronization of agents via the *Time* message and an new *Point* message which will give gaze positions not in the screen coordinates in pixels, which are not generic as the screen size could change, but in coordinates relative to the screen size. At last, implementation of Usybus on ZeroMQ, which has more implementations than Ivy, would be certainly useful to the community. Main UsyBus data types and reusable UsyBus agents will be downloadable on https://usybus.imag.fr.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and the participants for their involvement in this work. Jean Vanderdonckt acknowledges the support by the Wallonie Bruxelles International (WBI) under Grant No. 267168 (2016) and from the EU Pathfinder "Symbiotik" project. Francis Jambon acknowledges the Agence Nationale de la Recherche (ANR) which supports the TELEOS projet (ANR-06-BLAN-0243) and the Laboratoire d'Informatique de Grenoble (LIG) which supports the PILOTE2 and GELATI "Emergence" projects.

REFERENCES

[1] Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. 2019. Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'19)*. ACM, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/3314111.3319834

[2] Nathalie Aquino, Jean Vanderdonckt, and Oscar Pastor. 2010. Transformation Templates: Adding Flexibility to Model-Driven Engineering of User Interfaces. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 1195–1202. https://doi.org/10.1145/1774088.1774340

[3] Onur Asan and Yushi Yang. 2015. Using Eye Trackers for Usability Evaluation of Health Information Technology: A Systematic Literature Review. *JMIR Human Factors* 2, 1 (14 Apr 2015), e5. https://doi.org/10.2196/humanfactors.4062

[4] Roman Bednarik and Markku Tukiainen. 2006. An Eye-tracking Methodology for Characterizing Program Comprehension Processes. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'06)*. ACM, New York, NY, USA, 125–132. https://doi.org/10.1145/1117309.1117356

[5] Lars Torsten Berger, Andreas Schwager, Pascal Pagani, and Daniel M. Schneider. 2015. MIMO Power Line Communications. *IEEE Commun. Surv. Tutorials* 17, 1 (2015), 106–124. https://doi.org/10.1109/COMST.2014.2339276

[6] Tanja Blascheck, Kuno Kurzhals, Michael Raschke, M. Burch, Daniel Weiskopf, and Thomas Ertl. 2014. State-of-the-Art of Visualization for Eye Tracking Data. In *EuroVis - STARs*. The Eurographics Association. https://doi.org/10.2312/eurovisstar.20141173

[7] Tanja Blascheck, Kuno Kurzhals, Michael Raschke, M. Burch, Daniel Weiskopf, and Thomas Ertl. 2017. Visualization of Eye Tracking Data: A Taxonomy and Survey. *Computer Graphics Forum* 36, 8 (2017), 260–284. https://doi.org/10.1111/cgf.13079

[8] Tanja Blascheck, Kuno Kurzhals, Michael Raschke, Stefan Strohmaier, Daniel Weiskopf, and Thomas Ertl. 2016. AOI Hierarchies for Visual Exploration of Fixation Sequences. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'16)*. ACM, New York, NY, USA, 111–118. https://doi.org/10.1145/2857491.2857524

[9] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, Isabelle Provot, Jean Vanderdonckt, and Giovanni Zucchinetti. 1996. *Key Activities for a Development Methodology of Interactive Applications*. Springer London, London, 109–134. https://doi.org/10.1007/978-1-4471-1001-9_7

[10] Agnieszka Bojko. 2006. Using Eye Tracking to Compare Web Page Designs: A Case Study. *Journal of Usability Studies* 3, 1 (May 2006), 112–120. https://uxpajournal.org/wp-content/uploads/pdf/JUS_Bojko_May2006.pdf

[11] Sarah Bouzit, Gaëlle Calvary, Joëlle Coutaz, Denis Chêne, Eric Petit, and Jean Vanderdonckt. 2017. The PDA-LPA design space for user interface adaptation. In *11th International Conference on Research Challenges in Information Science (RCIS)*. 353–364. https://doi.org/10.1109/RCIS.2017.7956559

[12] Marcellin Buisson, Alexandre Bustico, Stéphane Chatty, Francois-Régis Colin, Yannick Jestin, Sébastien Maury, Christophe Mertz, and Philippe Truillet. 2002. Ivy: Un bus logiciel au service du développement de prototypes de systèmes interactifs. In *Conférence sur l'Interaction Homme-Machine (IHM'02)*. ACM, New York, NY, USA, 223–226. https://doi.org/10.1145/777005.777040

[13] Teresa Busjahn, Roman Bednarik, and Carsten Schulte. 2014. What Influences Dwell Time During Source Code Reading?: Analysis of Element Type and Frequency As Factors. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'14)*. ACM, New York, NY, USA, 335–338. https://doi.org/10.1145/2578153.2578211

[14] Shiwei Cheng and Anind K. Dey. 2019. I see, you design: user interface intelligent design system with eye tracking and interactive genetic algorithm. *CCF Trans. Pervasive Comput. Interact.* 1, 3 (2019), 224–236. https://doi.org/10.1007/s42486-019-00019-w

[15] Edward Cutrell and Zhiwei Guan. 2007. What Are You Looking for?: An Eye-tracking Study of Information Usage in Web Search. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'07)*. ACM, New York, NY, USA, 407–416. https://doi.org/10.1145/1240624.1240690

[16] Edwin S. Dalmaijer, Sebastiaan Mathôt, and Stefan Van der Stigchel. 2014. PyGaze: an open-source, cross-platform toolbox for minimal-effort programming of eyetracking experiments. *Behavior Research Methods* 49, 6 (2014), 913–921.

https://doi.org/10.3758/s13428-013-0422-2
[17] Soussan Djamasbi. 2014. Eye Tracking and Web Experience. *AIS Transactions on Human-Computer Interaction* 6, 2 (2014), 37–54. https://aisel.aisnet.org/thci/vol6/iss2/2
[18] Andrew T. Duchowski. 2007. *Eye Tracking Methodology: Theory and Practice.* Springer-Verlag, Berlin, Heidelberg.
[19] Anna Maria Feit, Lukas Vordemann, Seonwook Park, Caterina Berube, and Otmar Hilliges. 2020. Detecting Relevance during Decision-Making from Eye Movements for UI Adaptation. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'20).* ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3379155.3391321
[20] Upamanyu Ghose, Arvind A. Srinivasan, W. Paul Boyce, Hong Xu, and Eng Siong Chng. 2020. PyTrack: An end-to-end analysis toolkit for eye tracking. *Behavior Research Methods* 52 (2020), 2588–2603. https://doi.org/10.3758/s13428-020-01392-6
[21] Joseph H. Goldberg and Xerxes P. Kotval. 1999. Computer interface evaluation using eye movements: methods and constructs. *International Journal of Industrial Ergonomics* 24, 6 (1999), 631–645. https://doi.org/10.1016/S0169-8141(98)00068-7
[22] Gilleanes Thorwald Araujo Guedes and Rosa Maria Vicari. 2009. Applying AUML and UML 2 in the Multi-agent Systems Project. In *Advances in Conceptual Modeling - Challenging Perspectives.* Springer Berlin Heidelberg, Berlin, Heidelberg, 106–115.
[23] Prateek Hejmady and N. Hari Narayanan. 2012. Visual Attention Patterns During Program Debugging with an IDE. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'12).* ACM, New York, NY, USA, 197–200. https://doi.org/10.1145/2168556.2168592
[24] Tim Holmes and Johannes M. Zanker. 2012. Using an Oculomotor Signature as an Indicator of Aesthetic Preference. *i-Perception* 3, 7 (2012), 426–439. https://doi.org/10.1068/i0448aap
[25] ISO. 2019. *ISO/IEC 25010 - Software Quality Product Standard.* standard. International Standard Organization, Geneva. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0
[26] Mirjana Ivanović, Aleksandra Klašnja-Milićević, Jovana Ivković, and Marco Porta. 2017. Integration of Eye Tracking Technologies and Methods in an E-Learning System. In *8th Balkan Conference in Informatics (BCI'17).* ACM, New York, NY, USA, Article 29, 4 pages. https://doi.org/10.1145/3136273.3136278
[27] Robert J.K. Jacob and Keith S. Karn. 2003. Eye Tracking in Human-Computer Interaction and Usability Research: Ready to Deliver the Promises. In *The Mind's Eye.* North-Holland, Amsterdam, 573 – 605. https://doi.org/10.1016/B978-044451020-4/50031-1
[28] Robert J. K. Jacob. 1990. What You Look at is What You Get: Eye Movement-Based Interaction Techniques. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'90).* ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/97243.97246
[29] Clark William Jacobsohn and William August Hartman. 2017. *EyeSite: A Framework for Browser-Based Eye Tracking Studies.* Master's thesis. Worcester Polytechnic Institute, Worcester, Massachusetts, USA. https://digitalcommons.wpi.edu/mqp-all/2432/
[30] Francis Jambon. 2013. Analyse oculométrique "on-line" de prémices de fixations. In *7ème coloque de psychologie ergonomique (EPIQUE 2013).* Arpège Science Publishing, Bruxelles, 309–312. https://www.arpege-recherche.org/activites/colloques-epique/7e-colloque-epique
[31] Francis Jambon, Kévin Chappellet, and Gaëlle Calvary. 2013. Plasticité des Interfaces par Perception de l'Interaction Homme-Machine : illustration en oculométrie. 25ème conférence francophone sur l'Interaction Homme-Machine (IHM'13). https://hal.inria.fr/hal-00879529
[32] Francis Jambon and Vanda Luengo. 2012. Analyse oculométrique "on-line" avec zones d'intérêt dynamiques : application aux environnements d'apprentissage sur simulateur. In *Conférence sur Ergonomie et l'Interaction Homme-Machine (Ergo'IHM 2012).* Biarritz, France. https://doi.org/10.1145/2652574.2653401
[33] Víctor López-Jaquero, Jean Vanderdonckt, Francisco Montero Simarro, and Pascual González. 2007. Towards an Extended Model of User Interface Adaptation: The Isatine Framework. In *Proceedings of EIS 2007 Joint Working Conference on Engineering Interactive Systems, EHCI 2007-DSV-IS 2007-HCSE 2007.* (Salamanca, Spain) *(Lecture Notes in Computer Science, Vol. 4940),* Jan Gulliksen, Morten Borup Harning, Philippe A. Palanque, Gerrit C. van der Veer, and Janet Wesson (Eds.). Springer, 374–392. https://doi.org/10.1007/978-3-540-92698-6_23
[34] Patricia A. Carpenter Marcel Adam Just. 1976. The role of eye-fixation research in cognitive psychology. *Behavior Research Methods & Instrumentation* 8 (1976), 139–143. https://doi.org/10.3758/BF03201761
[35] Sandra Peery Marshall. 2008. Mental Alertness Level Determination. https://portal.unifiedpatents.com/patents/patent/US-7344251-B2 US Patent 7,344,251-B2.
[36] Unaizah Obaidellah, Michael Raschke, and Tanja Blascheck. 2019. Classification of Strategies for Solving Programming Problems Using AoI Sequence Analysis. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'19).* ACM, New York, NY, USA, Article 15, 9 pages. https://doi.org/10.1145/3314111.3319825

[37] Dario D. Salvucci and Joseph H. Goldberg. 2000. Identifying Fixations and Saccades in Eye-Tracking Protocols. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'00)*. ACM, New York, NY, USA, 71–78. https://doi.org/10.1145/355017.355028

[38] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 954–957. https://doi.org/10.1145/2786805.2803188

[39] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. 2012. An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'12)*. ACM, New York, NY, USA, 381–384. https://doi.org/10.1145/2168556.2168642

[40] Vaynee Sungeelee, Francis Jambon, and Philippe Mulhem. 2020. Proof of Concept and Evaluation of Eye Gaze Enhanced Relevance Feedback in Ecological Context. In *First Joint Conference of the Information Retrieval Communities in Europe (CIRCLE 2020) (CEUR Workshop Proceedings, Vol. 2621)*. http://ceur-ws.org/Vol-2621/CIRCLE20_04.pdf

[41] Jean Vanderdonckt, Mathieu Zen, and Radu-Daniel Vatavu. 2019. AB4Web: An On-Line A/B Tester for Comparing User Interface Design Alternatives. *Proc. ACM Hum.-Comput. Interact.* 3, EICS, Article 18 (jun 2019), 28 pages. https://doi.org/10.1145/3331160

[42] Sandeep Vidyapu, Vijaya Saradhi Vedula, and Samit Bhattacharya. 2019. Quantitative Visual Attention Prediction on Webpage Images Using Multiclass SVM. In *ACM Symposium on Eye Tracking Research & Applications (ETRA'19)*. ACM, New York, NY, USA, Article 90, 9 pages. https://doi.org/10.1145/3317960.3321614

[43] Adrian Voßkühler, Volkhard Nordmeier, Lars Kuchinke, and Arthur M. Jacobs. 2008. OGAMA (Open Gaze and Mouse Analyzer): Open-source software designed to analyze eye and mouse movements in slideshow study designs. *Behavior Research Methods* 40, 6 (November 2008), 1150–1162. https://doi.org/10.3758/BRM.40.4.1150

[44] Michael Wooldridge. 2009. *An Introduction to Multi-Agent Systems, 2nd Edition.* John Wiley & Sons, New York. https://www.wiley.com/en-us/An+Introduction+to+MultiAgent+Systems+2nd+Edition-p-9780470519462

[45] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: theory and practice. *The Knowledge Engineering Review* 10, 2 (1995), 115–152. https://doi.org/10.1017/S0269888900008122

A USYBUS DATA TYPES

This appendix details a selection of the main UsyBus data types contained in messages sent and received by agents on the exchange data bus.

Remark: In almost all data types, a "device" parameter distinguishes different transmitters providing the same type of data to guarantee unicity. In the context of eye-tracking, the "device" parameter represents the name of the surface monitored by an eye-tracker that the user is looking at. Usually, there is only one "device" per eye-tracker and per experiment, and its name is the DNS short name of the computer that host the eye-tracker controller. In some cases, there may be several "devices" per experiment (when there are several eye-tracker) or per eye-tracker (when an eye-tracker can monitor several surfaces).

A.1 Point

This type of message is usually sent by an eye-tracker controller. This data type describes the point the user is looking at on the screen for a defined timecode. The point is sent only if the coordinates are considered reliable by the eye-tracker. It consists of the gaze coordinates in pixels on the screen where the user's eyes are looking at. The type also has an optional fixed information specifying whether the gaze point belongs to a fixation.

```
type=eyetracking:point
Long tc                  – timecode (milliseconds since the epoch)
String device            – name of the device (alphanumeric)
Integer x                – x-axis coordinate of the point in the screen (pixels)
Integer y                – y-axis coordinate of the point in the screen (pixels)
Boolean fixed (optional) – fixation indication (true or false)
```

A.2 Pupils Sizes

This type of message is usually sent by an eye-tracker controller. This data type describes the diameters of the two pupils of the user for a defined timecode. The diameters values are expressed in the device-specific format of the eye-tracker system, and should be considered as relative to the distance between the user and the eye-tracker, unless otherwise indicated.

```
type=eyetracking:pupils
Long tc                  – timecode (milliseconds since the epoch)
String device            – name of the device (alphanumeric)
Double left              – left pupil diameter (device-specific format)
Double right             – right pupil diameter (device-specific format)
```

A.3 Time

This type of message is usually sent by an eye-tracker controller. This type represents the current time of the eye-tracker device real time clock. The value may be different than the host computer system time. This message may be used for synchronization purposes. The device parameter is not required since the real time clock is not linked with a specific screen.

```
type=eyetracking:time
Long tc                 – timecode (milliseconds since the epoch)
```

A.4 Device

This type of message is usually sent by the system that manages the screen seen by the user. This type describes the dimensions of the screen the user is looking at. This message is usually sent at the beginning of data collection to setup graphical monitoring systems.

```
type=eyetracking:device
String  device          – name of the device (alphanumeric)
Integer width           – width of the screen (pixels)
Integer height          – height of the screen (pixels)
```

A.5 Fixation

This type of message is usually sent by the eye movement analysis system. This type describes a fixation in the sense of the result of the analysis of eye movements. The type contains the position of the fixation on the screen, the mean radius, the maximum radius, and the duration of the fixation. Fixations are detected by a selectable ad-hoc algorithm. The timecode refers to the first point that belongs to the fixation.

```
type=eyetracking:fixation
Long tc                 – timecode (milliseconds since the epoch)
String  device          – name of the device (alphanumeric)
Integer x               – x-axis coordinate of the center of
                          the fixation in the screen (pixels)
Integer y               – y-axis coordinate of the center of
                          the fixation in the screen (pixels)
Integer meanradius      – mean radius of the fixation (pixels)
Integer maxradius       – maximum radius of the fixation (pixels)
Long duration           – duration of the fixation (milliseconds)
```

A.6 Zone

This type of message is usually sent by the target application that implements the user interface seen by the user. The zone represents an area of interest to analyse. Four types of zones are defined: rectangular, circular, elliptical, and punctual. Two virtual types of zones are defined: zone-to-remove and zone-to-remove-all to indicate the disappearance of one or all zones respectively. The type contains the position, the dimensions, and the name of the zone. The name of the zone must be unique for a specific device. The optional widget parameter can be used for the expanded eye-tracking monitor.

```
type=eyetracking:zone
Long tc                          – timecode (milliseconds since the epoch)
String device                    – name of the device (alphanumeric)
String type (for ZoneCircle, ZoneEllipse, ZonePoint, ZoneRectangle, ZoneToRemove, ZoneToRemoveAll)
                                 – type of the zone (enumeration)
String name (for all but ZoneToRemoveAll)
                                 – name of the zone (alphanumeric)
String widget (optional / for all but ZoneToRemoveAll)
                                 – widget type of the zone (alphanumeric)
Integer x (for ZoneCircle, ZoneEllipse, ZonePoint)
                                 – x-axis coordinate of the center (pixels)
Integer y (for ZoneCircle, ZoneEllipse, ZonePoint)
                                 – y-axis coordinate of the center (pixels)
Integer x1 (for ZoneRectangle)   – x-axis coordinate of the upper left corner (pixels)
Integer y1 (for ZoneRectangle)   – y-axis coordinate of the upper left corner (pixels)
Integer x2 (for ZoneRectangle)   – x-axis coordinate of the lower right corner (pixels)
Integer y2 (for ZoneRectangle)   – y-axis coordinate of the lower right corner (pixels)
Integer r (for ZoneCircle)       – radius of the circle (pixels)
Integer a (for ZoneEllipse)      – demi-width of the ellipse (pixels)
Integer b (for ZoneEllipse)      – demi-height of the ellipse (pixels)
```

A.7 Fixation in Zone

This message is usually sent by the eye movement analysis system. The fixation in zone type describes the zone that a user fixed during a certain amount of time.

```
type=eyetracking:fixinzone
Long tc                          – timecode (milliseconds since the epoch)
String device                    – name of the device (alphanumeric)
String name                      – name of the zone (alphanumeric)
Long duration                    – duration of fixation in the zone (milliseconds)
```

A.8 Cognitive Load

This type of message is usually sent by an eye-tracker controller. The type represents the data of the right and left Index of Cognitive Activity (ICA).

```
type=eyetracking:load
Long tc                  – timecode (milliseconds since the epoch)
String device            – name of the device (alphanumeric)
Double lICA              – left index of cognitive activity (numeric in 0..1)
Double rICA              – right index of cognitive activity (numeric in 0..1)
```

A.9 Task

This type of message is usually sent by the cognitive load monitor. The type represents the current user task.

```
type=eyetracking:task
Long tc                 – timecode (milliseconds since the epoch)
String device           – name of the device (alphanumeric)
String taskname         – name of the task (alphanumeric)
```

B HELLOWORLD APPLICATION EXAMPLE

The HelloWorld example is a simple application that uses the UsyBus framework to transmit a message between two UsyBus agents. A Sender agent sends text to the data exchange bus (*e.g.*, a firstname). A Receiver agent retrieves this text from the the data exchange bus, and displays it adding "Hello" before the text.

B.1 Architecture

The HelloWorld application is composed of two UsyBus agents: a Sender and a Receiver. The Sender agent sends a *Hello* message, that is received by the Receiver agent.

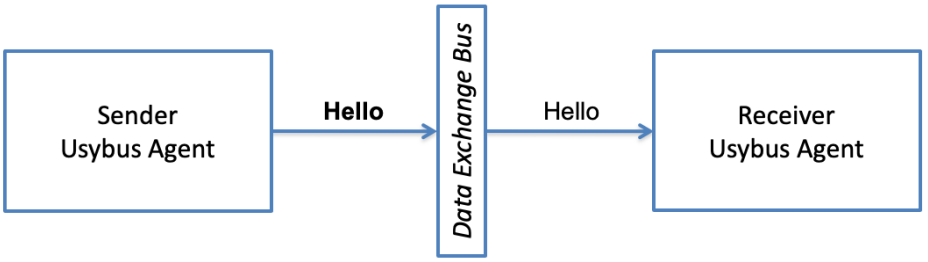

Fig. 27. HelloWorld multi-agent architecture based on UsyBus.

B.2 Data Type

HelloWorld application use the *Hello* data type, defined as follows.

```
type=hello
String text   – text to transmit (alphanumeric)
```

B.3 Source Code

Both agents use the `fr.imag.usybus` library to make a smooth link with the Ivy library.

The Sender agent sends text to the data exchange bus every second, and ten times:

```
package example.helloworld;

import fr.imag.usybus.*;

/**
 * HelloWorld Sender Agent.
 */
public class Sender {

    public static void main(String args[]) throws InterruptedException {
        // creates an Usybus agent named "Sender" (using Ivy messaging
            system)
        UsybusIvyAgent usybus = new UsybusIvyAgent("Sender");
        // connects the agent to the bus
        usybus.connect();
```

```
        // displays that the agent is connected
        System.out.println("Sender connected");
        // sends ten "Hello" messages
        for (int i = 0; i < 10; i++) {
            // waits 1 second
            Thread.sleep(1000);
            // creates an Usybus message
            UsybusMsg msg = new UsybusMsg();
            // sets the type of the message to "hello"
            msg.setType("hello");
            // adds String data to the message: variable="text" and
                value="world"
            msg.addDataAsString("text", "world");
            // sends the message
            usybus.sendMsg(msg);
            // displays that the message is sent
            System.out.println("Hello message with text=\"world\" sent");
        }
        // disconnects the agent from the bus
        usybus.disconnect();
        // displays that the agent is disconnected
        System.out.println("Sender disconnected");
    }

}
```

The Receiver agent receives text messages, and stops listening after 10 seconds:

```
package example.helloworld;

import fr.imag.usybus.*;

/**
 * HelloWorld Receiver Agent (extension of the UsybusIvyAgent class).
 */
public class Receiver extends UsybusIvyAgent {

    /**
     * Builder.
     * @param name UsyBus agent name
     */
    public Receiver(String name) {
        super(name);
    }

    /**
     * Receive UsyBus Messages of the type(s) binded.
     * @param msg UsyBus message
     */
    @Override
    public void msgReceived(UsybusMsg msg) {
```

```
        // tests if the type is correct and if the parameter exists in
            the datagram (these controls are useless as long as the
            types used are consistent)
        if (msg.getType().equals("hello") && msg.containsData("text")) {
            // displays "Hello" concatenated with the content of the
                "text" parameter received in the "hello" message
            System.out.println("Hello " + msg.getDataAsString("text"));
        }
    }

    public static void main(String args[]) throws InterruptedException {
        // creates an Usybus agent named "Receiver" (using Ivy messaging
            system)
        Receiver usybus = new Receiver("Receiver");
        // connects the agent to the bus
        usybus.connect();
        // displays that the agent is connected
        System.out.println("Receiver connected");
        // binds to messages of type "hello"
        usybus.bindType("hello");
        // waits 10 seconds (while listening to any messages)
        Thread.sleep(10000);
        // disconnects the agent from the bus
        usybus.disconnect();
        // displays that the agent is disconnected
        System.out.println("Receiver disconnected");
    }

}
```

B.4 Execution Output

Sender agent execution output:

```
    Sender connected
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Hello message with text="world" sent
    Sender disconnected
```

Receiver agent execution output:

```
Receiver connected
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Receiver disconnected
```

Received July 2021; revised September 2021; accepted October 2021