LittleAntispoof:
a multimodal face liveness detection system

Davide Quaranta, quaranta.1715742@studenti.uniroma1.it
Federico Cernera, cernera.1584227@studenti.uniroma1.it

Introduction

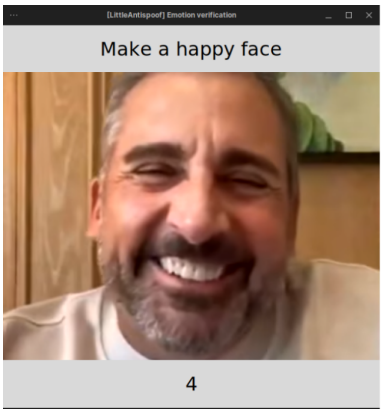
LittleAntispoof is a multimodal **face liveness detection module** that can be used in the context of face anti-spoofing.

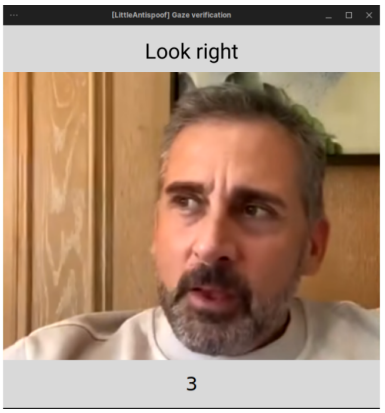
The system uses a challenge-response interaction style, composed of gaze verification, emotion verification, speech verification, and eye blinking check.

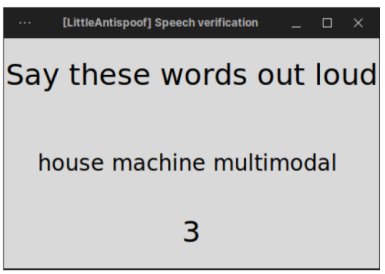
Specifically, recognition attempts that must be checked for liveness, will pass though the following phases, in a random order:

- **Gaze challenge**: the user is asked to look towards a certain direction, randomly chosen.
- **Emotion challenge**: the user is asked to provide an emotion-distorted face expression (e.g. happy face), randomly chosen.
- **Speech challenge**: the user is asked to pronounce a randomly chosen phrase.

During challenges, the user's **eye blinking rate** is also checked, in order to determine whether it is within a configured one.

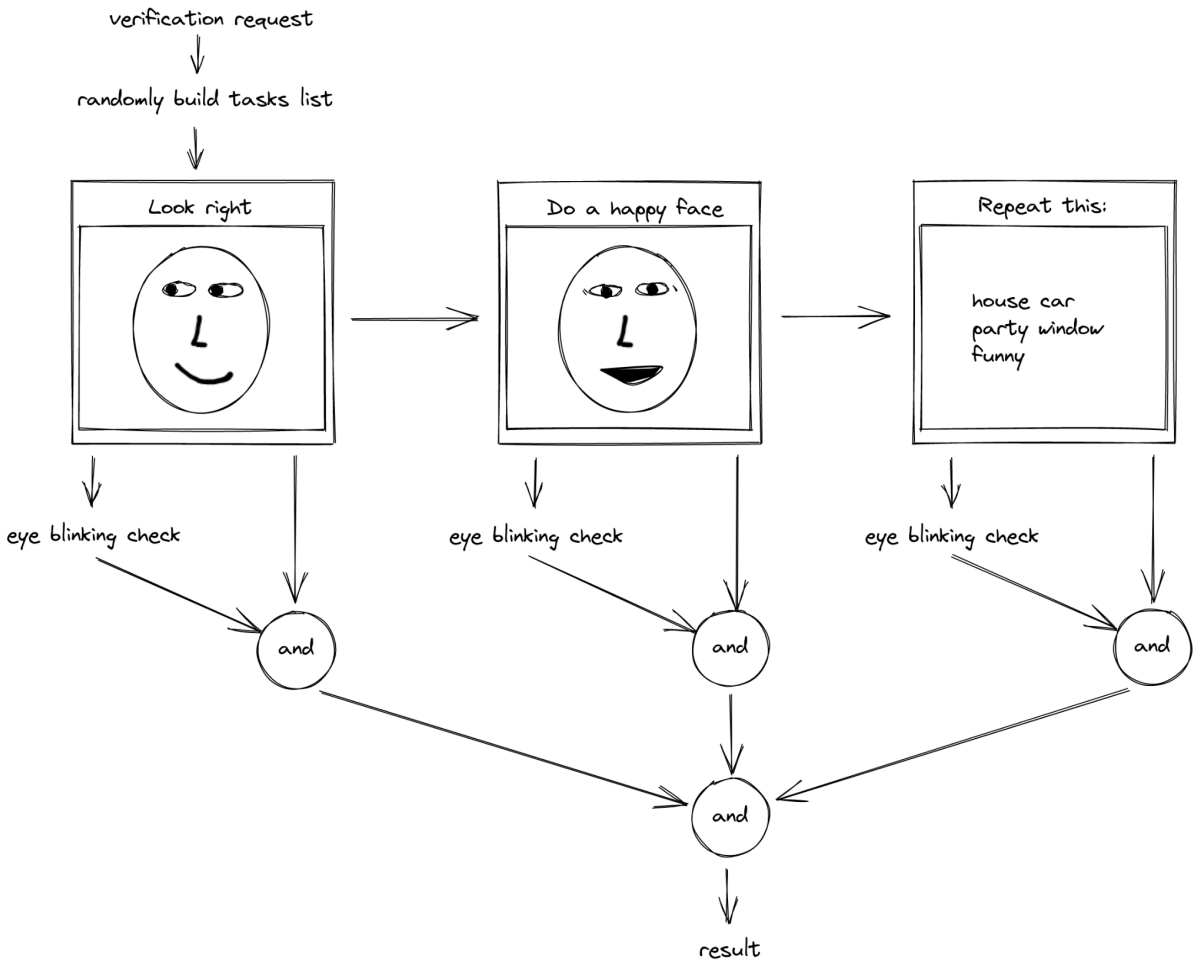






The system can be employed before or after a biometric recognition operation, to perform liveness detection. The result can then be combined with the recognition operation, to better determine whether to **accept** or **reject** the sample.

To summarize, the high-level behavior is described in the following figure:



System design

The system is an object oriented application written in **Python3**, built with the principles of modularity, extensibility and configurability.

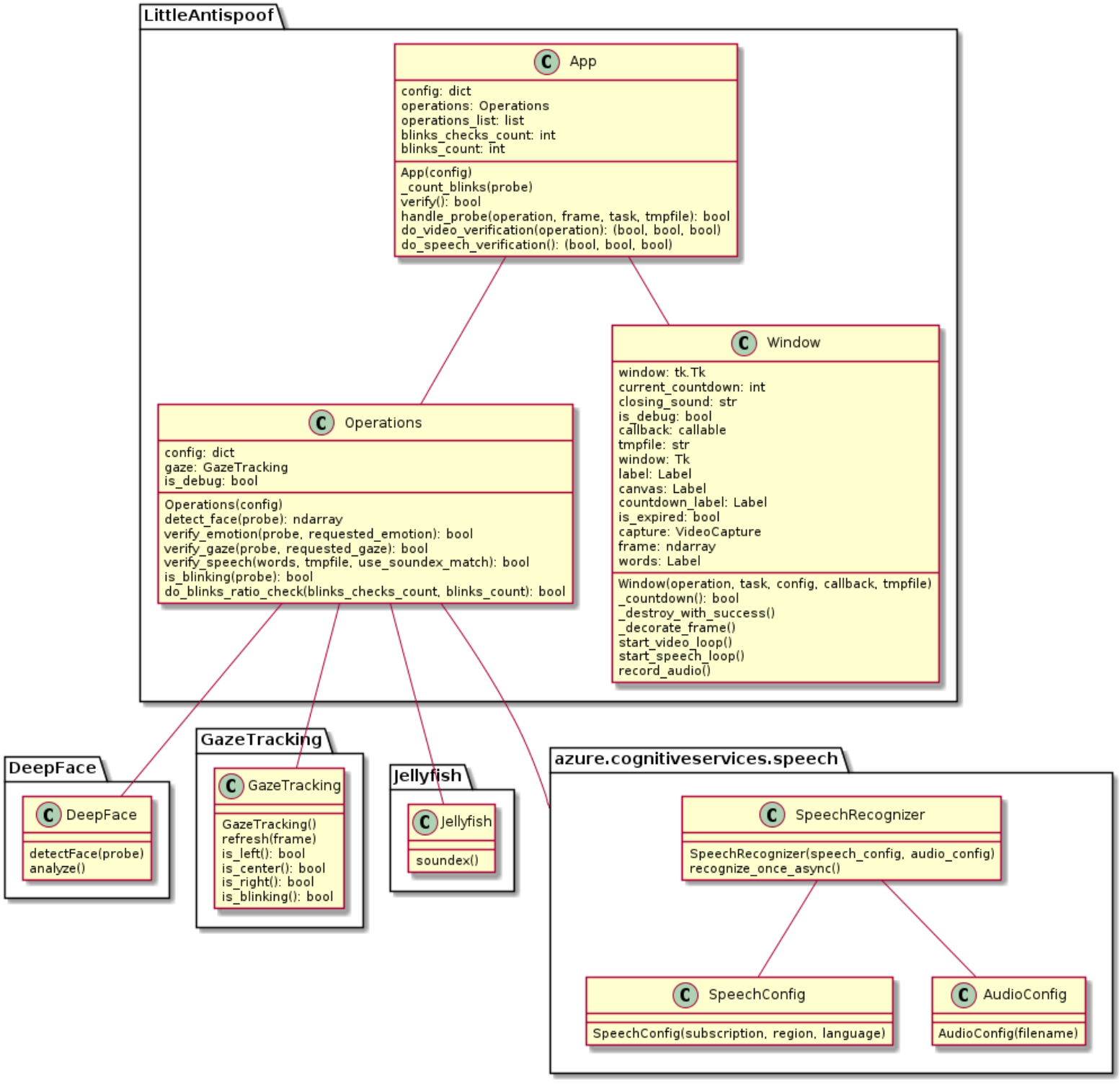
Project structure

The system is structured in the following main classes:

- **App**: main coordinator.
- **Operations**: handler for the various system operations.
- **Window**: handler for the GUI of the video and speech operations.

The GUI is built using the Tkinter library.

The following class diagram describes the project structure, also including the external modules DeepFace, Azure Cognitive Services, GazeTracking, and Jellyfish, needed respectively for emotion recognition, speech to text, gaze tracking and phonetic matching.



At startup time, the system reads a JSON configuration file in the project root.

```
{
   "face_detector": "dlib",
   "window_duration_secs": 5,
   "blinking": {
       "min_threshold": 0.2,
       "max_threshold": 0.6
   },
   "speech": {
       "words_to_display": 3,
       "use_soundex_match": true,
       "record_duration_secs": 5
   },
   "sounds": {
       "start": "./sounds/start.mp3",
       "camera_shot": "./sounds/camera_shot.wav",
       "mic_stop": "./sounds/mic_stop.wav"
   },
   "debug": true
}
```

Libraries

The LittleAntispoof class `Operations` interacts with external components (libraries) that realize the needed functionality, i.e. emotion recognition, gaze and blinking check, speech recognition, phonetic matching.

DeepFace

DeepFace[1] is a Python library that performs face recognition and face attributes analysis, including **emotion recognition**.

The following is a snippet of LittleAntispoof's code, as an example of a DeepFace emotion analysis call:

```
def verify_emotion(self, probe, requested_emotion: str) -> bool:
    """
    Returns True if the emotion extracted from the given probe matches
```

[1] DeepFace: https://github.com/serengil/deepface

```
    against the requested one.
    """
    result = DeepFace.analyze(
        probe,
        actions=["emotion"],
        detector_backend=self.config["emotion"]["detector_backend"],
        )
    return result["dominant_emotion"] == requested_emotion
```

As shown below, the call returns a dictionary containing an **emotion probability distribution**, along with the **dominant** (most probable) emotion and the image region containing the face.

```
{
    'emotion': {
        'angry': 2.0189900733421686e-06,
        'disgust': 6.294248737258371e-14,
        'fear': 5.569503523939416e-08,
        'happy': 99.99113678879361,
        'sad': 8.27286306586789e-06,
        'surprise': 7.136712091459289e-07,
        'neutral': 0.008858164601887143
    },
    'dominant_emotion': 'happy',
    'region': {
        'x': 245,
        'y': 76,
        'w': 221,
        'h': 221
    }
}
```

GazeTracking

GazeTracking[2] is a Python library that provides a **webcam-based eye tracking** module, and allows to extract the following information:

- Pupils coordinates.
- **Horizontal gaze** direction (left, center, right).
- Eye **closed/open** status.

[2] GazeTracking: https://github.com/antoinelame/GazeTracking

The library transparently handles pupils localization and **calibration**.

The needed information can easily be obtained with the following calls:

```
gaze = GazeTracking()
gaze.refresh(image)
gaze.is_left()
gaze.is_center()
gaze.is_right()
gaze.is_blinking()
```

The following code snippets are examples of how such information is used in the project.

```
def verify_gaze(self, probe, requested_gaze: str) -> bool:
    """
    Returns True if the given gaze in the given probe
    matches against the requested one
    """
    def __get_gaze_direction():
        if self.gaze.is_left():
            return TASK_GAZE_LEFT
        if self.gaze.is_center():
            return TASK_GAZE_CENTER
        if self.gaze.is_right():
            return TASK_GAZE_RIGHT

    self.gaze.refresh(probe)
    return requested_gaze == __get_gaze_direction()
```

The following method wraps the call to the library, to get whether eyes are open or closed.

```
def is_blinking(self, probe) -> bool:
    """
    Returns True if the eyes in the given probe are closed
    """
    self.gaze.refresh(probe)
    return self.gaze.is_blinking()
```

As shown in the last snippet, the library considers as a blink the simple status of closed eyes, hence in the project it is necessary to redefine the **blinking** concept and to detect an **event transition** from **open** to **closed** eyes.

To detect said transition, the eyes' status in the **previous frame** is considered with the help of the following function, that is called **for every frame**.

```
    def _count_blinks(self, probe):
        eyes_closed = self.operations.is_blinking(probe)
        if eyes_closed is None:
            if self.config["debug"]:
                print("Cannot locate eyes")
            return

        if eyes_closed and not self.were_previous_frame_eyes_closed:
            if self.config["debug"]:
                print("Blinking")
            self.blinks_count += 1

        self.were_previous_frame_eyes_closed = eyes_closed
```

Azure Cognitive Speech Services - STT

The speech-to-text functionality of the project is handled by Microsoft Azure's Speech to Text service[3], which can be used via its Python SDK.

The following code snippets show how said SDK is used in the project.

```
def verify_speech(self, words: str, tmpfile, use_soundex_match=True) ->
bool:
    def speech_to_text():
        """
        Returns the user utterance as text, from a recorded wav file.
        """
        speech_config = speechsdk.SpeechConfig(
            subscription=get_azure_api_key(), region=get_azure_region()
        )
        speech_config.speech_recognition_language = "en-US"
        audio_input = speechsdk.AudioConfig(filename=tmpfile)
```

[3]https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/

```
        speech_recognizer = speechsdk.SpeechRecognizer(
            speech_config=speech_config, audio_config=audio_input
        )

        result = speech_recognizer.recognize_once_async().get()
        final_result = result.text
        for char in [",", ".", "?", "!", ";"]:
            final_result = final_result.replace(char, "")
        return final_result.lower()
    # ...
    stt = speech_to_text()
    # ...
    return result
```

In the project, values for the Azure API Key (subscription IDs) and the region, are obtained from **environment variables**, respectively `AZURE_API_KEY` and `AZURE_REGION`, which can be conveniently set in the `.env` file in the project's root.

Jellyfish

Jellyfish is a library to perform **phonetic matching** of strings, which is useful to mitigate the effects of speech recognition errors.

In the project, Jellyfish is used to convert both the requested phrase and the user's speech into **Soundex codes**, to be matched against each other.

The following code snippet (extracted from the `speech_to_text()` function above) shows the process of converting a sentence to a list of soundex codes.

```
    def to_soundex(sentence: str) -> list:
    """Converts the given string to a list of Soundex codes"""
        codes = []
        for word in sentence.split(" "):
            codes.append(soundex(word))
    return codes
```

Then matching can be done on the results of `to_soundex()`.

```
    stt = speech_to_text()
```

```
    result = (
        to_soundex(words) == to_soundex(stt) if use_soundex_match
        else words == stt
    )
```

As an example, the output of `to_soundex(str)` is:

```
>>> to_soundex("this is an example of the soundex match")
['T200', 'I200', 'A500', 'E251', 'O100', 'T000', 'S532', 'M320']
```

As further examples, `"got to do"`, `"got too doo"` and `"got two doo"` produce the same soundex sequence `['G300', 'T000', 'D000']`.

Other libraries

The following utility libraries were also used in the project:

- **PIL**[4]: image manipulation library, used in conjunction with Tkinter.
- **OpenCV**[5]: computer vision library, used in the project to display the webcam feed.
- **playsound**[6]: sound reproduction library.
- **sounddevice**[7]: audio signal recording library.
- **python-dotenv**[8]: environment variables handling library.

Multimodality

Within the borders of this system, the act of looking in a specific direction or to reproduce a specific facial expression, are a form of **gesture**.

Hence, the **multimodal** nature of the project is determined by the use of multiple modalities, namely **gesture interaction** and **speech interaction**.

Context

Our system is assumed to be used in a familiar environment, where the user is mostly in control of its surroundings; specifically we define the various environments that compose the context in this way:

---

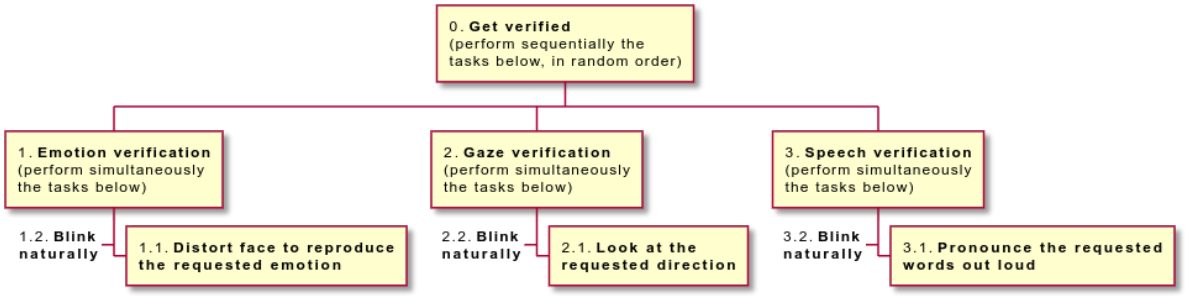
[4] https://github.com/python-pillow/Pillow
[5] https://opencv.org/
[6] https://github.com/TaylorSMarks/playsound
[7] https://github.com/spatialaudio/python-sounddevice/
[8] https://github.com/theskumar/python-dotenv

- **Computational environment**:
  - A video acquisition device is installed and accessible.
  - An audio acquisition device is installed and accessible.
  - An Internet connection is established.
  - The network must be able to reach Microsoft Azure.
- **Physical environment**:
  - The amount of illumination is sufficiently high to allow distinguishing facial landmarks of the user.
  - The amount of noise in the room is sufficiently low to allow words to be caught with acceptable quality by a recording device.
  - The user is isolated from other people.
- **User environment**:
  - The user is aware of the system's existence.
  - The user is cooperative towards the system.
  - The user can read on screen.
  - The user can speak.
  - The user can produce facial expressions.
  - The user's eyes are well visible.

Interaction design

The interaction with our system is composed of a single macro-task called "Get verified", which is related to the user goal of **being verified by the system**.

The following **HTA** (**Hierarchical Task Analysis**) diagram describes the tasks decomposition.



The macro-task is decomposed into a sequence of **sub-tasks**, one for each **verification step**, namely emotion, gaze, speech. These sub-tasks are executed in **sequence**, with a **random** order each time.

Consequently, these subtasks have further **decompositions** indexed x.1 and x.2, that are instead executed **concurrently**.

Furthermore, the sub-tasks related to **blinking** (x.2) are drawn without a box to remark the fact that they are **implicit and transparent to the user**. Namely, they are needed to fulfill the upper task, but the user doesn't know about it.

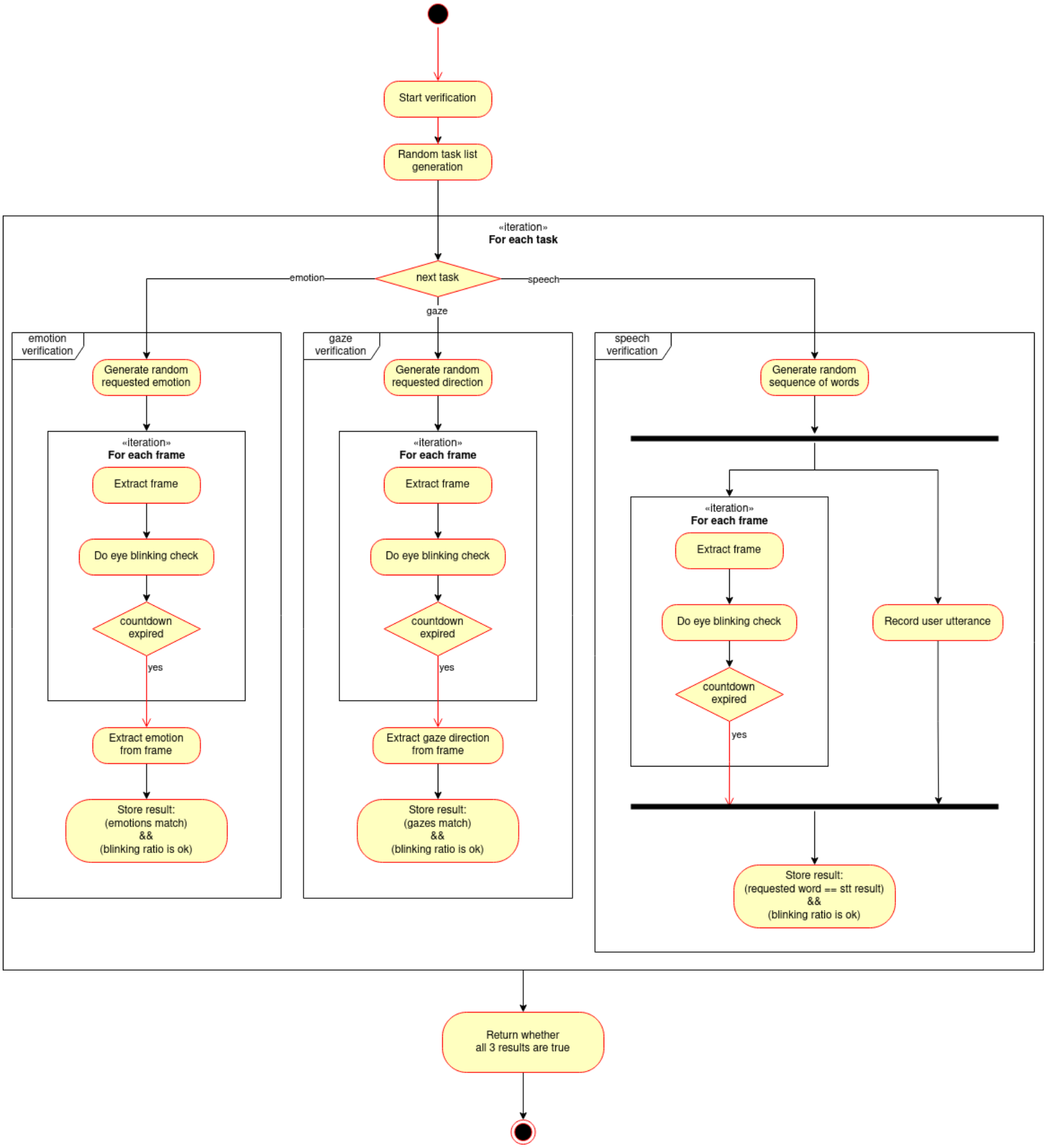
Fusion

Since the project uses two **modalities** (i.e. gestures and speech), a multimodal fusion strategy is needed to merge the received information.

Specifically, our project employs **decision-level fusion** between the modalities described in the following table.

Information type	**Modality**
Emotion	Gestural
Gaze	Gestural
Speech	Speech
Blinking	Gestural (implicit)

The activity diagram on the next page illustrates how the single tasks work and how the decision-level fusion is applied.


Start verification
Random task list generation
«iteration»
For each task
next task
emotion
gaze
speech
emotion verification
Generate random requested emotion
«iteration»
For each frame
Extract frame
Do eye blinking check
countdown expired
yes
Extract emotion from frame
Store result:
(emotions match)
&&
(blinking ratio is ok)
gaze verification
Generate random requested direction
«iteration»
For each frame
Extract frame
Do eye blinking check
countdown expired
yes
Extract gaze direction from frame
Store result:
(gazes match)
&&
(blinking ratio is ok)
speech verification
Generate random sequence of words
«iteration»
For each frame
Extract frame
Do eye blinking check
countdown expired
yes
Record user utterance
Store result:
(requested word == stt result)
&&
(blinking ratio is ok)
Return whether
all 3 results are true

To summarize the previous diagram:

- The list of tasks is randomly generated.
- The result of each verification task is the logical `AND` between:
  - The result of the detection operation itself.
  - The result of the eye-blinking check.
- The final end result is the logical `AND` between the results of every task.

Regarding the **speech verification** task, as introduced in the System design section, the project employs **phonetic matching** to reduce the effects of possible speech recognition errors.

Fission

The main **output information** that must be organized (multimodal fission) is:

- The event of a task starting.
- The specific user challenge (“call to action”) for the current task.
- Updates on the current task.
- The event of a task ending.

Since the project is a **liveness detection module**, it is not necessary–nor advisable–to provide feedback on the success status of each task, since that information could be exploited to run a **side-channel attack**.

Thus, **success feedback** must be considered only for the **final result**, consisting of the combination of the results of each challenge.

Instead, information that helps the user to correctly conduct the task can be derived from the reasoning described in the following subsections, which directly correspond to above bullet points.

Initial user focus: start event

When the liveness detection module starts, the user may not be focused on the screen, hence it is needed to catch their **attention** with a **redundant modality**.

For this reason, a **sound** is reproduced to catch the user's attention.

Challenge guidance: call to action

After a task has started and the user’s attention is caught, the user must be provided with a short textual description of the challenge.

Since the possible challenges may require the user to remove their focus on the screen (and consequently to the UI), an automatic **countdown** timer is employed to acquire the needed frame, allowing the user not to press any button, but to **focus on fulfilling the challenge**.

For example, when the gaze task “look right” starts, the user has a number of seconds (determined by the countdown timer) to look at the requested direction, without worrying about what is on the screen.

Challenge guidance: updates and task type

When the task starts, the user also can **deduce** some information by looking at the **kind of interface** that is presented on screen; from the user’s point of view:

- “The window contains my webcam feed: I’m requested to do a gesture”.
- “The window contains a phrase: I’m requested to pronounce it”.

Moreover, looking at the countdown timer update, the user can also deduce that the interface automatically handles the frame acquisition, hence they can focus on fulfilling the requested challenge.

Repeated user focus: end event

Since the user may not look at the screen while executing a challenge, their attention must be caught again, to communicate that the task is finished and they should **focus** on what is the **next challenge** (if any).

For this reason, **non-speech audio** is again employed:

- After a video challenge, a camera shot **auditory icon** is reproduced.
- After a speech challenge, an **earcon** is reproduced.

Conclusion

As a conclusion, a **multimodal face liveness detection module** has been designed and developed, using a mixture of **emotion recognition**, **gaze detection** and **speech recognition**, condensed into the modalities of **gestural interaction** and **speech interaction**.

Background **blinking checks** are used to further strengthen the security of the module, by counting the number of blinks in a certain time window and checking whether the ratio between blinks and time is within the configured range.

The **HTA** methodology has been followed to design the interaction, by decomposing the verification task into sub-tasks.

In **multimodal fusion**, the system performs a decision-level fusion, by aggregating the results of each verification sub-task; furthermore, the list of verification steps is randomly generated at each attempt.

The system also employs **phonetic matching** to mitigate the effects of speech recognition errors.

In **fission**, a mixture of on-screen text and **non-speech audio** is employed to assist the user by giving partial **feedback** on the task status and evolution, and to direct the **user's focus** as needed.

Future work

The modular nature of the system allows it to be easily modified to introduce further extensions, as described in this section.

Redundancy

Redundant modalities can be introduced to strengthen the interaction and generally to offer more **flexibility**.
For example, text-to-speech can be employed to read the task out loud; it would be redundant with regards to the standard video modality.

Support for other gestures

The kinds of tasks can be extended to introduce other modalities or gesture styles, such as hand gestures.

Support for complex challenges

Similarly to the previous point, the system can be extended to support more complex tasks, such as:

- Voice solvable CAPTCHA.
- Grammatical challenge (e.g. pronounce the subject of a sentence).