SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of
Informatics, Systemistics and COmmunications

PhD program: Informatics Cycle XXIX

Software Architectures For Embedded Systems Supporting Assisted Living

Surname: Mobilio Name: Marco

Registration number 701812

Tutor: Prof. Giuseppe Vizzari

Supervisor: Prof. Daniela Micucci

Coordinator: Prof. Stefania Bandini

ACADEMIC YEAR 2016/2017

"To my Grandfather"

Abstract

In coming decades, population is set to become slightly smaller in more developed countries, but much older. This increase results in a growing need for supports (human or technological) that enables the older population to perform daily activities. This originated an increasing interest in Ambient Assisted Living (AAL), which encompasses technological solutions supporting elderly people in their daily life at their homes.

The structure of an AAL system generally includes a layer in charge of acquiring data from the field, and a layer in charge of realising the application logic. For example, a fall detection system, acquires both accelerometer and acoustic data from the field, and exploits them to detect fall by relying on a machine learning technique.

Usually, AAL system are implemented as vertical solutions in which often there is not a clear separation between the two main layers. This rises several issues, which include at least a poor reuse of the system components since their responsibilities overlap, and a scarce liability to software evolution mostly because data is strongly coupled with its source, thus changing the source requires modifying the application logic too.

To promote reusability and evolution, an AAL system should keep accurately separated issues related to acquisition from those related to reasoning. It follows that data, once acquired, should be completely decoupled from its source. This allows to change the physical characteristics of the sources of information without affecting the application logic layer. Moreover, the acquisition layer, should be structured so that the basic acquisition mechanisms (triggering sources at specified frequencies, and distributing the acquired data) should be kept separated from the part of the software that

interacts whit the specific source (i.e., the software driver). This allows to reuse the basic mechanisms and to program the drivers for the needed sensors only. If a new or different sensor is required, it suffices to add/change the sensor driver and to properly configure the basic mechanisms so that the change can actually implemented.

The aim of this work is to propose a novel approach to the design of the acquisition layer that overcomes the limitation of traditional solutions. The approach consists of two different sets of architectural abstractions:

Time Driven Sensor Hub (TDSH) is a set of architectural abstraction for developing timed acquisition systems that are easily configurable for what concerns both the type of the sensors needed and their acquisition frequencies.

Subjective sPaces Architecture for Contextualising hEterogeneous Sources (SPACES) is a set of architectural abstractions aimed at representing sensors measurements that are independent from the sensors characteristics. Such set can reduce the effort for data fusion and interpretation, moreover it enforces both the reuse of existing infrastructure and the openness of the sensing layer by providing a common framework for representing sensors readings.

The final result of this work consists in two concrete designs and implementations that reify the TDSH and SPACES models. A test scenario has been considered to contextualise the usefulness of the proposed approaches and to test the actual correctness of each component.

The example scenario is built upon the case of fall detection, an application case studied in order to be aware of peculiarities of the chosen domain. The example system is based on the proposed sets of architectural abstraction and exploits an accelerometer and a linear microphonic array to perform fall detection.

Contents

List of Figures

List of Tables

Introduction

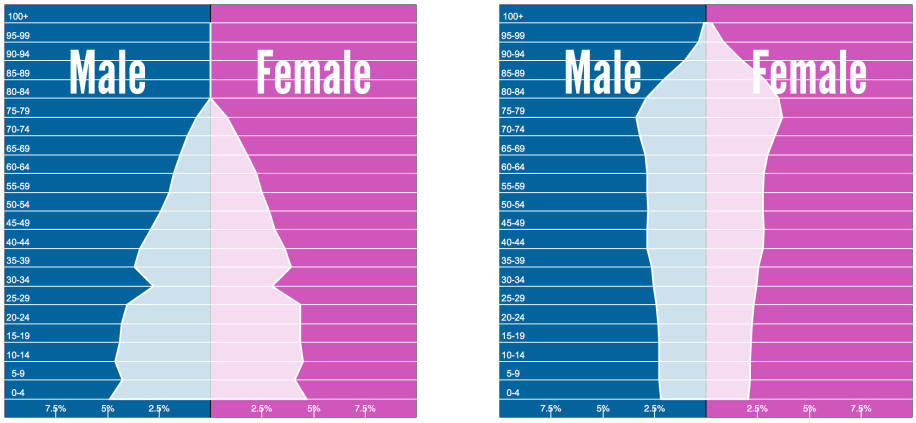
Motivations

The rise in life expectancy is one of the great achievements of the twentieth century. As an example life expectancy in 1950 was 65 years in the more developed countries and 42 in the less developed regions. By 2010-2015, it is estimated to be 78 years and 68 years respectively. This is a still running trend, as life expectancy is projected to reach 83 years in developed regions and 75 years in less developed regions by 2045-2050 [92]. Figure 1 clearly shows how this trend reflects in the population pyramid of Italy, from 1950 to 2050. It is clearly visible how in 1950 (Figure 1a) the vast majority of the population was under 35 years old, while it is expected to by over 60 in 2050 (Figure 1b).

Another factor that is contributing to the increase of the population average age is a long-term downtrend in fertility that is being experienced by the most developed countries, especially in Europe [42]. As a result natural population growth rates are in decline or even decrease. This decline in fertility is happening after the phenomenon known as *baby booming*, which is defined as the steep increase in fertility during the years after the second World War (usually confined between 1946 and 1964); the progress of baby-boomers toward retirement age will represent a substantial increase and fast in the proportion of old people.

Moreover, the net immigration rates which could theoretically offset the decline in the working-age population, have remained generally low in most European countries. Those demographic trends cause *population ageing*, which rises a number of issues:

- The decrease of the working-age population results in decline in human capital, which could reduce productivity.
- Pension and social insurance systems can become heavily burdened.



(a) 1950, population: 46.111.000 (b) 2050, population: 56.512.000

Figure 1: Italy Population Pyramid.

- A growing number of elderly will require long-term health care services. Considering constant the current use rates, the number of people requiring such services will double by 2040 [12], increasing the related public spending.

- Population in need of care services will increase much faster than the working age population, this could result in the impossibility of providing the needed services even in the case of financial stability. The ratio between the number of dependents (people younger than 15 or older than 64) and the working age population (those ages between 15 and 64) is known as the *Age Dependency Ratio*, Figure 2 shows the dependency ratio of Italy and Japan (another fast ageing country) compared with the World average from 1960 to 2014: from the picture, it is clearly visible the increasing size of the elderly compared to the global population.

The oncoming shortage of caregivers and the strong desire of the great majority of older adults to live in their own homes and communities [36] originated a still increasing interest in what has been defined as *Ambient Assisted Living* [52]. AAL encompasses technical systems to support people in their daily routines to allow an independent and safe lifestyle as long as possible. Often AAL solutions focus on the needs of special interest groups other than elderly, such as people with disabilities or people with temporarily need of assistance [41].

The spectrum of AAL systems is very wide, for example there are platforms for social inclusion that aim at keeping the users in touch with their caregivers and family members; proposals focus on keeping the users physically and mentally active proposing daily activities and allowing them to keep track of their progress [80].

AAL systems usually rely on information from the environment in order to

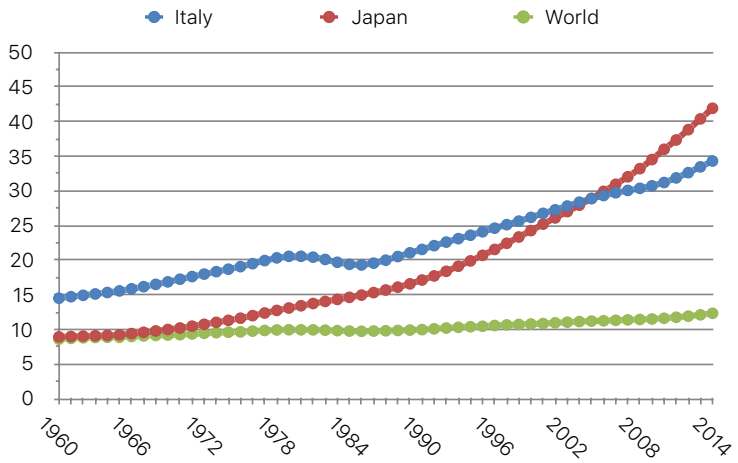


Figure 2: Dependency Ratio of japan, Italy, and World average from 1960 to 2014 (World DataBank Data).

properly work. All the systems that rely on sensors and actuators in order to perform any kind of activity recognition and act on the environment to provide feedback falls within this category.

One of the main issues in this kind of systems is *sensor heterogeneity*: to perform their elaboration applications often require different kinds of data and therefore different sensors. Moreover, sensor information is not enough to allow applications to make meaningful inferences. Information about the spatial position of sensed data is often a key component in AAL applications and Ambient Intelligence in general. For example, a movement or a sound coming from a location known to be forbidden may be used in a home surveillance system (AmI) or a loud noise might trigger an alarm as symptom of a fall (AAL), but it would make sense only to trigger the alarm in the case of sound being from near the floor (e.g. someone or something falling) instead of mid air (e.g. hands clapping).

Another peculiarity of AAL systems is that different sensors can be used to obtain the same high level information. For example movements can be detected from cameras, from microphones, or presence sensors (PIRs). Moreover there are applications in which sensors types and positions differ from user to user in order to be effective. For example systems for monitoring of physically impaired people: pressure sensors may be used to determine if a user is spending to much time on a static position on the wheelchair or applying to much pressure. The number and position of pressure sensors should be determined individually for each user as everyone has different issues and peculiarities. Customisation and easy configurability are therefore key factors for these systems.

In most cases domain applications have no direct interest in knowing the exact source of the information or how it has been inferred. In the majority of available solutions however, they have to consider low level information about

the physical sources in order to interpret data or add the spatial information. These kind of behaviours, while allowing a precise control and knowledge of the data, reduce the reusability of software, and they are not resilient to changes in the sensors configuration [88].

The following solutions are good candidate to achieve reusability and evolution:

- Separation between acquisition dynamics and sensors configurations. This allows both to have simultaneously different type of sensors with different frequency controlled by the same acquisition node, and to ease the customisation of the sensors according to the application and the users.
- Data exploited by the application logic should be as decoupled by its physical source so that a change of the sensor does not imply an upgrade of the application logic also. Moreover, since location and time of the observed events may help the application logic to make its inferences, acquired data should be placed in in temporal and space context.

Contributions

The aim of this work is to propose a novel approach to the design of the acquisition layer that overcomes the limitation of traditional solutions. The approach consists of two different sets of architectural abstractions:

1. *Time Driven Sensor Hub* (TDSH) is a set of architectural abstraction for developing timed acquisition systems that are easily configurable both in terms of sampling rates and kind of sensors. The derived framework consists in a set of architectural abstractions that allow time-related aspects to be explicitly treated as first-class objects at the application level. Both the temporal behaviour of an application and the way the application deals with information placed in a temporal context can be modelled by means of such abstractions, thus narrowing the semantic gap between specification and implementation. Moreover, TDSH carefully separates behavioural policies from implementation details improving portability and simplifying the realisation of adaptive systems.
2. *Subjective sPaces Architecture for Contextualising hEterogeneous Sources* (SPACES), a set of architectural abstractions aimed at representing sensors measurements that are independent from the sensors technology. Such set can reduce the effort for data fusion and interpretation, moreover it enforces both the reuse of existing infrastructure and the openness of the sensing layer by providing a common framework for representing sensors readings. The abstractions rely on the concepts of space. Data is localised both in a positioning and in a measurement space that are subjective with respect to the entity that is observing the data. Mapping functions allow data to be mapped into different spaces so that different entities relying on different spaces can reason on data.

Both the proposals have been implemented in order to test their feasibility. Moreover a test scenario is provided to contextualise the usefulness of the proposed approaches and to test the actual correctness of each component.

Outline

The rest of the document is organised as follows:

- Chapter 1 gives a brief overview of the state of the art in Ambient Assisted Living and its enabling technologies. It also mentions some systems aimed at data acquisition and some available systems.
- Chapter 2 presents a general overview of the proposed abstractions, dividing it in two main activities.
- Chapter 3 explains the TDSH model, along with its implementation
- Chapter 4 presents the SPACES model, describing its main concepts and usage.
- Chapter 5 draws some conclusions and presents future development about the proposed models.

Acknowledgements

First of all I would like to thank my mighty mentor and supervisor Prof. Daniela Micucci for being the best guide for the academic world I could hope to find.

I also wish to thank Prof. Stefania Bandini and Prof. Hiroko Kudo for giving me suggestions, reviewing my thesis, and making my visiting time in Tokyo possible; Prof. Toshi Kato for the hospitality at Chuo University, it was an amazing experience; Prof. Carlos Medrano, my other reviewer, for the useful comments and insights.

A mention is due to my colleagues and friends at SAL, both from the past and currents.

Finally I would like to thank Prof. Leonardo Mariani, for giving me the opportunity to continue my journey in the academic world.

CHAPTER 1

State of the Art

This chapter gives an overview of the state of the art in Ambient Assisted Living systems. In particular, Section 1.1 gives a more specific contextualisation of AAL inside the field of Ambient Intelligence and Assisted Living. Data acquisition systems in general terms are also described, it is noteworthy how only the acquisition and representation of sensor data are the focus of this work, with long-term storage, reasoning and actuation being out of scope.

1.1 Ambient Assisted Living Systems

The oncoming shortage of caregivers, along with the strong desire of the great majority of older adults to live in their own homes and communities instead of institutional settings [36] originated a still increasing interest in what has been defined as Ambient Assisted Living (AAL) [52].

AAL encompasses technical systems to support people in their daily routines to allow an independent and safe lifestyle as long as possible. Often AAL solutions focus on the needs of special interest groups other than elderly, such as people with disabilities or people with temporarily need of assistance [41]. The main goal of AAL has been defined in [52] as the application of Ambient Intelligence (AmI) technology to enable people with specific demands. AmI is considered as a (relatively) new research area for distributed, non-intrusive, and intelligent software systems [79]. Along the years many different definitions of AmI have been proposed and in [27] the different features of AmI systems expressed by the different definitions have been classified as in Table 1.1 and are the followings: Sensitive (S), Responsive (R), Adaptive (A), Transparent (T), Ubiquitous (U), and Intelligent (I). Sensitivity, Responsiveness and Adaptivity are concepts that closely relates AmI with *Context-Aware Systems*, a term

Table 1.1: AmI features captured by different definitions

Definition	**S**	**R**	**A**	**T**	**U**	**I**
A developing technology that will increasingly make our everyday environment sensitive and responsive to our presence [4].	X	X				
A potential future in which we will be surrounded by intelligent objects and in which the environment will recognise the presence of persons and will respond to it in an undetectable manner [34].	X	X		X	X	
It is an environment where digital technology senses what people want, through interconnected and personalized interfaces embedded, invisibly, around us [74].	X	X	X		X	X
A vision of future daily life... contains the assumption that intelligent technology should disappear into our environment to bring humans an easy and entertaining life [29].		X		X	X	
A new research area for distributed, non-intrusive, and intelligent software systems[79].				X		X
In an AmI environment people are surrounded with networks of embedded intelligent devices that can sense their state, anticipate, and perhaps adapt to their needs [94].	X		X	X	X	X

introduced in [82]: these systems are aware of the users presence (sensitivity), interact with him (responsiveness) and can adapt based on the context (adaptivity). On the other hand, transparency and ubiquity are derived from the concepts of *disappearing computer* [97] and *ubiquitous computing* [96], both introduced by Mark Weiser in 1991 and 1993 respectively.

A similar feature analysis has been performed by Acampora et al. in [8], in which the characteristics of an AmI system are:

- Context awareness, it exploits contextual and situational information.
- Personalisation, it is tailored to the different needs of each user.
- Anticipation, it can understand specific needs without active actions from the user.
- Adaptivity, it is able to adapt to the changing needs of individuals
- Ubiquity, it is seamlessly integrated with the environment.
- Transparency, it does not stay in the way of the user, it is part of the background.

While here "intelligence" is not stated as a feature, is intended as a key aspect: in particular, by exploiting Artificial Intelligence (AI), AmI systems can be *more sensitive, responsive, adaptive, and ubiquitous* [27, 8]. In [79], specifically, AmI is seen as a research area that stands at the intersection between AI and Software Engineering (SE). AmI algorithms perceive the state of the environment and users with sensors, reasons about the data using a variety of AI techniques and acts upon the environment using actuators in order to achieve their goals.

1.1.1 AAL Stakeholders

Understanding who are the main stakeholders in AAL and their needs is of crucial importance to design and develop useful AmI systems for Assisted Living (AL). Four different classes of stakeholders have been identified in [3]:

1. The primary stakeholders are the elderly users and their informal caregivers (mainly their families).
2. The secondary stakeholders group includes service providers for the primary stakeholders.
3. The Tertiary stakeholders are the organisations supplying goods and services (i.e. the AAL technologies producers).
4. The quaternary stakeholders include the policy makers, insurance companies and the other organisations that analyse the economic and legal context of AAL.

The main needs of the elderly are about:

- Social Inclusion: they should be able to contribute to the society, maintains connections with their social networks and in general to reduce loneliness, insecurity, vulnerability, and isolation, especially in rural areas.
- Quality of Living: they should be supported in their home environment in order to reduce the risk of accidents, to enable early detection of developing illnesses, and to provide prompt help in case of accidents. Moreover those illness already present, such as chronic diseases should be managed at home, rather than requiring hospitalisation.
- Human Rights: They should be able to maintain an adequate purchasing power to satisfy their primary needs. Moreover the avoidance of any form of maltreatment is a crucial point for people with dementia living with caregivers.

Informal caregivers, on the other hand, are mainly family members with no professional experience in long-term care services. They performs about 60% of the care requests and are usually unpaid. Without proper training or support they can suffer from physical and psychological issues and they caregiving work

may not be adequate. For these reasons, they need to be supported with policies and training as well as to be equipped with the right technological tools in order to provide optimal services and assisting in critical decisions.

1.2 Enabling Technologies of AAL Systems

AAL systems usually rely on the *sense-act/interact* loop depicted in Figure 1.1.

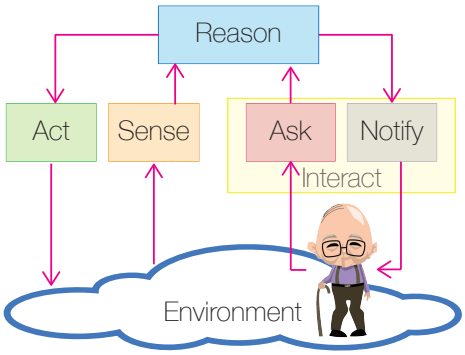


Figure 1.1: Capabilities in AAL systems.

The *Sensing* and the *Asking* activities capture respectively information from the environment and wanted from the users. *Reasoning* is in charge of interpreting captured data to act on the environment and on the user respectively through the *Acting* and the *Notifing* capabilities. The user can be considered as part of the environment itself: information about him can be obtained through Q&A or observation capabilities. Finally, in order to cooperate, each activity relies on *Communicating* technologies depicted as pink arrows in Figure 1.1.

1.2.1 Sensing

Sensing is the fundamental capability of an AAL system because sensors capture information about the environments and the people who inhabit it. Sensors are usually enriched with processing and communication capabilities. Such sensors are commonly called *smart sensors*, which can be seen as a special case of *smart objects*, that is, autonomous cyber-physical objects augmented with sensing (or actuating), processing, storing, and networking capabilities [39]. In AAL systems sensors are generally divided in two main categories: *wearable* and *environmental*.

Wearable Sensors.

Wearable sensors are positioned directly or indirectly on the human body. They usually monitor the physiological state of a person and her/his position and body

movements. Concerning the person's physical state, a wide range of parameters can be obtained from different sensors, for example:

- *Tympanic, skin, oral, and rectal temperatures* are obtained by thermistors.
- *Blood pressure* is sensed through sphygmomanometer cuff [71].
- *Carbon dioxide* is commonly measured by a capnograph.
- *Oxygen saturation* is acquired by devices that rely on pulse oximetry.
- *Heart's electrical activity* is measured with a electrocardiography.
- *Blood chemistry* is usually sensed by means of chemical sensors.

Person's position and movements are commonly exploited in order to perform ADLs (Activities of Daily Living) recognition and classification [61] and, more recently, fall detection [63, 56]. The most common monitored parameters are:

- *Outdoor position* is generally acquired via GPS (Global Positioning System) devices by the resection process using the distances measured to satellites.
- *Detection and identification of a person* are generally obtained by Frequency IDentification (RFID).
- *Body position and movement* are normally obtained by tri-axial accelerometers, magnetometers, and angular rate sensors.

Environmental Sensors.

Environmental sensors are embedded into the environment. They typically detect conditions that are descriptive of the environment or interactions between users and the environment. Research in this specific field is usually divided between *video-based* and *non-video-based* solutions.

Video-Based AAL Solutions. Vision-based solutions for AAL applications (VAAL) is a trending topic mainly due to the high versatility of cameras. The most explored areas are activity recognition in the rehabilitation and health care [22], and fall detection [81, 68]. A noteworthy innovative approach is in exploiting video technology to recognise and monitor physiological data. The main concern over the adoption of VAAL is the loss of privacy [22]. Moreover, those solutions must be accepted by potential users and their families, who may have concerns even in applications that claim to ensure privacy [100].

Non-Video-Based AAL Solutions. Sensors in this category usually have only a few parameters they can monitor, reason for which they are often combined together. Some examples of sensed parameters are:

- *Ambient light* is usually measured with sensors based on photodiodes.
- *Room temperature* is acquired as body temperature, thus using thermistors.
- *Humidity* is usually sensed by a Relative Humidity (RH) sensor.
- *Movement* and *presence* are usually sensed by Passive Infrared Sensors (PIR).
- *Door/window/cabinet open/closed* is usually obtained by a magnetic proximity switch based on reed elements.
- *Pressure*, intended as the force applied toward a surface, is obtained by force-sensing resistors that can be easily attached to flat surfaces such as chairs.
- *Environmental sounds* are sensed through microphones. The more widely adopted are Electret, which are specific kind of capacitor microphones that do not need a constant source of electrical charge to operate. Microphones can be used as presence sensor (like PIRs) or to achieve acoustic source localisation [72, 50]. Localising the source of a sound can be used to perform a more precise indoor positioning or for fall detection [73, 77, 56].
- *Odours* provide a lot of information about the surrounding environment. In recent years, many researchers have focused on developing olfactory sensors, able to capture and distinguish odours [31].

Environmental sensors overcome the main issue of wearable sensors by not requiring the users to always wear them. However, they have their own issues (apart privacy and acceptability problems): their price is higher, their require installation (and, thus, related cost), and they are fixed on their location, thus operating as long as the user is at home.

Trends in Sensor Technology.

Since wearable sensors loose their functionalities if not worn, the research trends are toward size and weight reduction, durability, and waterproofing. *Microelectromechanical Systems* (MEMS, but it is also known as micro-machines in Japan or Micro Systems Technology (MST) in Europe) is an innovative technology consisting in miniaturising mechanical and electromechanical elements using micro-fabrication techniques. Miniaturisation has also enabled *ingestible sensors* and *implantable sensors*, mostly used in professional medical environments. Ingestible sensors are systems integrated into ingested devices such as pills. They are conceived to be powered by the body and communicate through

the tissue. These sensors can monitor ingested food, weight, and various physiological parameters, but also body position and activity, thus favouring users sustaining healthy habits and clinicians providing more effective healthcare services [78]. Implantable sensors are used in post-surgery: once implanted they can monitor and transmit data about the load, strain, pressure, and temperature of the healing site of surgery.

1.2.2 Reasoning

Reasoning is the process of converting data acquired from the field to meaningful information, which may have different meaning at multiple levels of interpretation (e.g., 12 o'clock (noon) may mean 12:00, mid-day, day time and so on), depending on the personal context of the user. *Personal Context* is defined as user specific context information: parts of the environment (e.g. things, services, and other persons) accessed by the user; the physiological state (e.g. pulse, blood pressure, weight) and psychological state (e.g. mood and stress); the tasks that are being performed; the social aspects of the current user (e.g. friends, neutrals, co-workers, relatives); the spatio-temporal aspects of the other context components from the user point of view [69]. The main properties related to reasoning are: *data collection and processing*; *activity recognition, modelling, and prediction*; *decision support*; *spatio-temporal reasoning.* Different reasoning modules exploiting different properties can be combined in a single application. Artificial Intelligence (AI) can help in obtaining better performing modules and thus to be able to produce more useful applications.

Data collection and processing.

Data acquired via the sense activity is usually easy to collect and process, however the amount of such data is a challenge especially if audio and visual information is included. Being able to obtain and integrate information from different kinds of sensors and sources is crucial to make AAL systems able to recognise events and conditions and, thus, to identify contexts and status. This skill is called *sensor data fusion* and is defined as the process of combining data to refine state estimates and predictions [86].

Activity recognition, modelling, and prediction.

Reasoning technologies in AAL should be able to understand the contexts and the current status not only by using static rules and patters, but also dynamic and reactive models that take into consideration complex information (e.g., behaviour models of users). Moreover, they should be able to extract relevant information (data mining) and update the same models (learning machine). Specifically, AAL systems need to have capabilities such as: reinforcement learning (i.e., learning from the world observations), learning to learn (i.e., learning from previous experiences), developmental learning (i.e., learning from the world

exploration), and e-Learning (i.e., learning from the Web and information technology) [3].

One of the main contribution that reasoning algorithms offer is the ability to recognise user activities. Different methods are available to recognise activities [8]: *template matching techniques* [17], *generative approaches* [20, 89, 98], *decision trees* [62], *discriminative approaches* [58, 38, 63].

Models of the user behaviours and the recognition of activities are fundamental for predicting probable statuses and context outcomes. This property is necessary both for anticipating possible negative events and conditions, thus acting in order to avoid them, and for predicting desires of the users, thus increasing their satisfaction.

Although recognising normal activities has a key role in health applications, abnormal events are very important too, as they usually indicate a crisis or an abrupt change in regimen that is associated with health issues. Likewise normal activities, abnormal activities can be recognised by classifiers, which usually require to be trained with datasets containing examples of the activities to be recognised. However, datasets containing activities related to critical situations (such as heart disfunction or falls) are rarely available. For these reasons, anomaly detection in AAL is receiving an increasing interest [77, 23, 63].

Decision support.

Decision Support System (DSS) is a general term for any computer application that supports enhanced decision making. DSSs have been widely adopted in healthcare, assisting physicians and professionals in general by analysing patients data [51].

Spatio-temporal reasoning.

Being able to reason on spatial and temporal dimensions is a key element for understanding the current situation. For example, a smart house system is able to recognise if someone turns on a cooker and leaves it unattended for more than 10 minutes; if this happens the system takes action by autonomously turning off the cooker and/or warning the user [27]. Thus, a number of proposal have been made in order to enable spatio-temporal reasoning in AAL contexts [14, 40, 64].

1.2.3 Interacting

Interaction is a well studied area under the umbrella of the Human Computer Interactions (HCI) and it encompasses all kinds of tool, both software and hardware, that allow the interaction process between the user and the system [7]. When designing an AAL system, attention must be put in the interacting activity because it has been pointed out that AAL systems will go unused if they are difficult or unnatural to use for the residents, especially the elderly.

The HCI may be *explicit* or *implicit*. Explicit HCI (eHCI) is used explicitly by the user who ask the system for something. This kind of interaction is in

direct contrast with the idea of invisible computing, disappearing interfaces, and ambient intelligence in general. eHCI always require some sort of dialog between a user and the system and this dialog brings the computer to the centre of the user's activity.

Implicit HCI (iHCI) tries to reduce the gap between natural interaction and HCI by including implicit elements into the communication: the system acquires *implicit input* (i.e., human actions and behaviours done to achieve a goal, not primarily regarded as interaction with a computer) and may present *implicit output* (i.e., output from a computer that is not directly related to an explicit input and that is seamlessly integrated with the environment and the task of the user) [83]. The basic idea is that the system can perceive users' interaction with the physical environment, and, thus, can anticipate the goals of the user.

Towards a Natural Interaction.

The analysis of the key issues in interaction and communication between humans offers a starting point toward new forms of HCIs. Three concepts have been identified as crucial toward better interactions:

- *Shared knowledge.* In interactions between humans a common knowledge base is essential; it is usually extensive but not explicitly mentioned. Any communication between humans takes some sort of common knowledge for granted and it usually includes a complete world and language model, which is obvious for humans but very hard to grasp formally.

- *Communication errors and recovery.* Communication is almost never error free. Conversations may include small misunderstandings and ambiguities, however in a normal dialog these issues are solved by the speakers through reiteration. In human conversations is therefore normal to relay on the ability of recognise and resolve communication errors. However, in interactive computer systems that are invisible, such abilities are less trivial.

- *Situation and context.* The meaning of the words as well as the way the human communication is carried out are heavily influenced by the context (i.e., the environment and the situation that lead to communication), which provides a common ground that generates implicit conventions.

Comparing the way in which people interact to the way people interact with machines, it becomes clear that HCIs are still at their early stages. What humans expect from interactions is dependent on the situation, which is one of the concepts on which the field of Context Awareness Computing is based [6].

Interaction in the AAL domain.

As mentioned at the beginning of this section, one of the key aspects in the success of any technological solution is its usability and acceptability according to end-user perspectives [3].

This is particularly true in AAL because most of the current and near future end-users of any AAL system are individuals with low to none affinity for technology. In order to develop successful interfaces for AAL services, designers should act accordingly to usability and acceptability criteria. Among all the theories, the most important are the Technology Acceptance Model [33], the Unified Theory of Acceptance and Use of Technology [95], and the Usability Theory [2].

1.2.4 Acting

Adding acting capabilities to an AAL system can be seen as obtaining the equivalent of a Closed Loop Control System in Control Theory, although the parameters affected by the actuation may not always be monitored by sensors and not every sensed parameter may be influenced by the actuations.

While sensors are required to understand and monitor the physical world, *actuators* are those mechanical objects that act on the physical world as a consequence of a software system action.

The number of different available sensors greatly outnumbers the number of actuators. However a few key actuators are sufficient to build a large number of complex smart objects.

The most common and simple actuators are already present in most of the homes, but almost always they are standalone systems. For example, indoor illumination and air conditioning (AC) systems. First attempts into making illumination systems more context aware have been achieved by coupling lightbulbs with motion sensors (PIRs): this way the lights do not require any explicit interaction in order to be switched on or off, but the movement of the user is an implicit input that cause the lights switching.

Actuators.

As mentioned, there is a small set of common actuators that are used as building blocks in AAL systems. Some examples are:

- *Relays* are usually electromechanical devices acting as remote switches that can be activated by a software system through a low-power signal.
- *MOSFETs* (Metal-Oxide-Semiconductor Field-Effect Transistors) are transistors and serve as switches. Compared to relays, MOSFETs are usually very small and some of them can switch almost 10 orders of magnitude faster then relays. However, magnetic fields, static electricity, and heat can easily broke them. They are usually employed to operate in low amperage situations (e.g., to switch on/off led lights or motors and servos).
- *Lights* have been among the first actuators included in AmI system. Modern lights for AAL usually support dimmer facilities, provide different light colours, and include a small micro controller handling communication. Most of the modern lightning solutions are based on Light Emitting

Diodes (LEDs) that can come to full brightness without need for a warm-up time.

- *Motors* commonly used are the DC (Direct Current) ones. Their are used in garage doors, curtains, or wheel chairs. A DC motor is a device that converts electrical energy into mechanical energy. In order to increase precision, stepper motors are usually adopted. Another highly used class of electric motors are servo motors, which are electric motors that can push or rotate an object with great precision. Servo motors are commonly adopted for precise, small movements that may require high torque.

- *Screens and speakers* provide feedback or information by transforming electrical data into physical phenomenons, light emissions, and sound waves respectively.

- *Haptic feedback engines* date back to 1968 [87], but only in recent studies they have been consistently considered in AmI solutions. Haptic Interfaces are used to provide tactile feedback (skin perception of temperature and pressure). It is a technology that complements visual and audio channels [3]. Force and positional feedback is considered as the next step of haptic interfaces for Virtual Reality, as they can also provide information on strength, weight, force, and shape.

1.2.5 Communication

Communicating capabilities are key aspects of AALs, since they are usually made up of distributed devices cooperating to provide the desired services. Three different types of networks are considered in AAL systems:

- *WANs* are employed whenever an AAL system needs to transmit information outside the system. Today solutions usually exploit an Internet connection obtained through one of the different providers available. With the increasing number of devices connected to the Internet, identification and addressing have been the most studied issues, which resulted in IPv6.

- *LANs* are used within home systems. They count different classes of technologies, such as cabled connections, powerline communications or wireless LAN (WLAN). Home automation often exploits dedicated buses, which means that gateways must be considered in order to put home automation systems in communication with the rest of the AAL structure.

- *BANs* derives from the widespread use of wearable devices [8, 49]. In a BAN sensors and actuators (mostly haptic, sound, or visual) are attached on clothes or directly on the body and less frequently implanted under the skin. BANs are characterised by three communication layers: intra-BAN (communication within the BAN), inter-BAN (connection between body sensors and Access Points), and beyond-BAN (streaming body sensor data to metropolitan areas, for example, to remote database where the

users' profiles and medical histories are stored and made accessible to professional caregivers.)

1.3 Data Acquisition Systems

In this context Data Acquisition Systems are those systems aimed at acquiring data from physical sensors. They can be seen as a specific branch of Cyber-Physical Systems (CPS). While there is no formal definition, CPSs are defined as transformative technologies for managing interconnected systems between its physical assets and computational capabilities [16, 55].

1.3.1 Challenges of Data Acquisition Systems

In the branch of CPSs that deals with sensor information, common solutions feature a vertical integration of all the concerned steps, from the acquisition from the physical sources, through the manipulation of the data, up to the fruition of information. While this kind of vertical solutions offer a tight control over the final applications at the expenses of hardware and middleware flexibility, as well as software reuse in general. In order to avoid such architectural pitfalls, as an example, [88] presents a prototype architecture for CPSs. The proposal is based on a series of observations about the interaction between the human society and the physical world. From these considerations, a list of basic properties that future CPSs should have:

- Global Reference Time. It should be accepted by every component, including human users, physical devices and software logic;
- Event/Information Driven. In this context *Events* represents the raw facts acquired by sensors (or perceived by users) and named *Sensor Events*, or actions performed by actuators (or, again, users) defined as *Actuator Events*. Information represents the abstraction of the physical world obtained through event processing;
- Dynamic Confidence. This concept is strictly related to the property of global reference time (in terms of an acquisition time consistent at system level) and *Lifespan*, which determines how much time has to pass before the confidence of an event/information drops to zero. Confidence and confidence fading equations determine events and information confidence and how it fades over time. These values can vary widely depending on the application domain, but the concepts should be applicable to every CPSs. A principle related to the event/information confidence is *Trustworthiness*, defined as the amount of trust that a receiver component has with regard to a specific event/information source.
- Publish/Subscribe Scheme. The idea that the best approach as communication scheme derives from the fact that it is currently adopted in the human society. By using this mechanism each CPS module acts as a

human being that only subscribes to interesting events/information and publishes new information when necessary.

- Semantic Control Laws. With this term context awareness and user customisation is intended. With the abstraction of the real-time physical world proposed, system behaviours related to the environment context according to user defined conditions and scenarios can be controlled.

- New Networking Techniques. In order to support future CPSs, new techniques for data transmission that support and are optimised for the publish/subscribe scheme and timing synchronisation. Moreover, they propose that if a network is able to determine that the confidence of an information drops to zero, it could be unnecessary to continue forwarding that information.

This list of properties highlights a number of open research challenges such as how a global reference time can be provided in a large scale heterogenous system or how the event/information model should be formalised.

Moreover, in [54], the issue of reliability is presented. Embedded systems have required levels of reliability and predictability rather than general-purpose computing. However the physical world they interact with is not entirely predictable, which means that they must be robust to unexpected conditions and adaptable to subsystem failures. The approach proposed is that at any level of abstraction components should be predictable and reliable if technology allows it. In case reliability and predictability are not feasible at the current level, the next abstraction level should compensate with robustness. Another issue exposed in [54] is the lack of the concept of timing in the semantics of standard programming languages. As an example, in [99] the authors claim that "it is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time". Hiding timing properties has been a common practice for computer scientist, however in an embedded system, computations interact directly with the physical world where time related concepts may not be avoided.

1.3.2 Available Systems and approaches

A unified and accepted approach to overcome the presented issues, while satisfying the mentioned properties is still missing. There are a great variety of acquisition systems proposed in the literature, however they are usually big distributed systems, such as the ones presented in [28, 11, 9] where the acquisition part of the system is usually just presented as an "acquisition node" without any information on how the software on such node is structured. Even smaller systems, usually more focused on single tasks such as fall detection or human activities recognition suffer from the same issue of non-disclosure about the acquisition nodes platforms, they are usually just mentioned as present and the focus is on the task of choice or the communication of data[5, 25, 76]. As some

of these proposals exploit smartphones as acquisition nodes, they usually are just extracted through dedicated applications [23, 30, 85].

There are however a number of different proposals focusing on subsets of the discussed issues, some of them are here presented.

Time Specific solutions

Giotto, as an example, represents a programming language extended to include timed semantics. Its main goal is the separation between logical correctness (functionality and timing) and physical realisation (mapping and scheduling), while remaining platform independent. It has been proposed in [45] and it is specifically aimed for embedded control applications.

In the Giotto model, the period invocation of tasks, reading of sensor values, and writing of actuator values are triggered by real-time with the result that a Giotto program does not specify in any way when tasks are scheduled, the only assurance provided by the Giotto compiler is the respect of the logical semantics (functionality and timings).

While the Giotto model assure that tasks timings are respected, such timings are not exposed as a first class object, thus they are not dynamically available at the application level.

Sensors and actuators representation

A different approach is presented in [26]. Here timing aspects are not treated, but the proposed system focus on the abstraction of the physical world and a data driven approach. Specifically, the concept of *Virtual Node* is presented: a virtual node is a programming abstraction aimed at simplify the development of decentralised applications; data acquired by a set of sensors can be aggregated and elaborated according to an application-provided function and treated as the reading of a single *Virtual Sensor*. Similarly, a *Virtual Actuator* acts as an aggregation point for the distribution of commands to a set of actual actuator nodes. Virtual nodes allow the application developer to focus on the application logic by treating a number of different nodes as one, thus not having to deal with the complexity of communication and data management.

The concepts proposed in [26] are contextualised in the case of Wireless Sensor Networks (WSN), but the underlying ideas stand also for different topologies. The main issues in this approach are based on the fact that the aggregation function is integrated within the virtual nodes, but are application dependant, which means that the high level developer has to deal with the lower levels of the system where it would be ideal, in order to improve reuse, maintainability and modularity, to have as few levels as possible that are domain dependant. Moreover the architecture of virtual nodes is not exposed, thus making it impossible to reason about the scalability and maintainability of the lower levels of the system.

A slightly different definition of Virtual Sensor (VS) is given in [60]: here a VS is defined as a product of spatial, temporal and/or thematic transformation

of data (either raw data or information from other VSs). A VS behaves like a real sensor, in terms of emitting timed data, but it may include newly defined thematic concepts or observations that are not available from the raw sensor point of view. An ontology for data streams and the concept of VS are also proposed. The Virtual Sensor Ontology include geographical information about the sensors in order to group VSs into different presentation and analysis layers based on their position.

Data representation

Another example is [75], where the authors propose a layered architecture that provides the low-level software, the middleware, and the upper-level services with detailed specifications of the involved sensors. In this approach sensors are well modelled, but their knowledge is distributed throughout all the system. In [32] the issues of management of large amount of data acquired from sensors are considered: the proposed approach consists in transforming sensor data in what authors call a *set of observations* that are meaningful for the applications. Lower levels embed semantics that is strictly related to the specific application. This lead to scarce reusability as the same abstraction rules for a specific sensor may not be applicable in different contexts.

Finally, database approach is growing interest. Indeed, the database approach allows heterogeneous applications to access the same sensor data via declarative queries. This kind of solutions may resolve data heterogeneity at the application level, but there still persists the issue of sensor data management, since most of the existing solutions suppose homogeneous sensors generating data according to the same format [44].

The described proposals are valid regardless of the application domain, since the focus of this work is on Ambient Assisted Living, an in-depth review of some enabling technologies, such as sensors and actuators is done in Section 1.2 within the description of the AAL domain, where also systems that cover the whole pipeline, from acquisition to fruition of sensor data are presented.

1.4 AAL Systems and Platforms

1.4.1 Evolution of AAL Technology

There are three generations of technologies supporting AAL [43, 18].

First generation solutions requires users to wear a device, generally equipped with a button that the user can press in order to alert call centers, informal caregivers (family members), or emergency services. A reduction of the stress levels among the users and the caregivers, the reduction of hospital admissions, and the delayed transfers to long-term care facilities are some of the benefits achieved [84]. The limitations are mainly related to the responsive-only nature of the systems: if the user is physically harmed or mentally incapacitated, she/he may not be able to trigger the alarm. Moreover, highly risk situations such as night wandering may occur without the device being worn.

Second generation solutions usually feature a proactive behaviour. They are able to autonomously detect emergency situations, such as falls [68], or environmental hazards, such as gas leaks [70]. As they do not require an interaction with the user, these systems are especially suitable for older adults with normal cognitive ageing or mild cognitive impairment [70, 18]. The main drawback is the obtrusiveness of the employed devices.

Third generation solutions are the most advanced and exploit recent ICT advancements. Third generation solutions are not only able to detect and report problems, but proactively try to prevent problems and emergency situations. Prevention can be achieved by two different activities: the first is the monitoring of the user's vital signs, and of any eventual change in his mobility and activity patterns, thus predicting ongoing changes in health status; the second activity is aimed at limiting the exposure of the user to high risk situations on the basis of actions performed and by using actuators.

Fall detection systems represents a good example of three stages of evolution of AAL systems: early proposals were passive and relied on the user actions; contemporary solutions are autonomous and proactively detect falls; finally, most innovative approaches are going toward falls prediction and avoidance.

Falls represent a major health risk that impacts the quality of life of elderly. Roughly 30% of the over 65 population falls at least once per year, the rate rapidly increases with age and among people affected by Alzheimer's disease. Fallers not able to get up by themselves and that lay for an extended period will more likely require hospitalisation and face higher dying risks [63].

The factors that impact the risk of falls have been classified in two categories: *intrinsic* and *extrinsic* risk factors. Intrinsic risk factors include age, low mobility, bone fragility, poor balance, chronic diseases, cognitive and dementia problems, Parkinson disease, sight problems, use of drugs that can affect the mind, incorrect lifestyle (inactivity, use of alcohol, and obesity), and previous falls. Extrinsic risk factors are usually related to incorrect use of shoes and clothes as well as drugs cocktails. Finally some environmental risk factors related to indoor falls have been identified as slipping floors, stairs, and the need to reach high objects. Only 8% of people without any of the risk factors fell, compared to 19% of people with one risk factors, 32% of people with two, 60% of people with three, and 78% with four or more risk factors [90]. In order to promptly detect and notify falls, most common technological solutions exploit wearables accelerometers embedded in smartphones [63, 85, 30] or ad-hoc devices [56, 48]. Most of the proposals use domain knowledge algorithms, usually based on empirically defined thresholds. More advanced solutions exploit machine learning techniques, with most of them requiring fall data in order to properly train the classifiers. Since real fall data are quite difficult to achieve, those solutions rely on simulated falls. However, simulated falls are not truly representative of actual falls [53]. Thus, Micucci et al. [63] evaluate the efficacy of anomaly detectors trained on ADL data only. Their findings suggest that prior understanding of fall patterns is not required.

1.4.2 Existing AAL Platforms

A number of platforms have been proposed in the literature, one of the first and more general purpose AAL projects was *CASAS*, that stands for Center for Advanced Studies in Adaptive Systems. Its goal is to design a smart home kit that is lightweight, extendable, and with a set of key capabilities [28]. In CASAS environments as intelligent agents, whose status (and of their residents) is perceived using several environmental sensors. Actions are taken using controllers with the aim of improving comfort, safety, and/or productivity of the residents. A three layered architecture characterizes CASAS: the Physical layer deals with sense and act activities, the Middleware layer manages communication exploiting the publish/subscribe paradigm, and the Application layer hosts applications that reason on the data provided by the middleware.

Other solutions are more directly focused toward the phenomenon of the ageing population and therefore to the elderly.

As an example, the iNtelligent Integrated Network For Aged people (NINFA) is a project focused on the users wellness. The aim is to build a service platform suited for elder people whose user interface allows to deliver at home different services, such as user supervision, communication and interaction among users for social inclusion, exergame delivering [80], and general monitoring of the wellness [67]. To allow an early diagnose, discourse and conversation analysis is applied to monitor verbal behaviour of people affected by different types of disorders (e.g., aphasia, traumatic brain injury, dementia). Moreover, to perform motor/cognitive analysis, the system deliveries a set of custom designed exergames via HCIs suitable for elderly or motor impaired patients. Another solution focused on prevention of age-related issues is *ROBOCARE*. The ROBOCARE approach comprises sensors, robots, and other intelligent agents that collaborate to support users. ROBOCARE is an example of a branch of AAL solutions that are exploring the advantages and challenges of integrating assistive and social robots within the systems. Specifically, it is based on a mobile robot unit, an environmental stereo-camera, and a wearable activity monitoring module. Based on the observations obtained by the camera and the wearable unit, the system applies automated reasoning to determine if the user activities fall within predefined and valid patterns. Such patterns are defined by caregivers also considering the user's medical state [24].

There are also other solutions, that aim to support users with specific needs, regardless of their age. As an example, the BackHome project is focused on designing, implementing, and validating person centred solutions to end users with functional diversity. The project aims at studying how brain-neural computer interfaces and other assistive technologies can help professionals, users, and their families in the transition from hospitalisation to home care. BackHome main goal is to help end users to accomplish goals that are otherwise impossible, difficult, or create dependence on a carer [15]. The outcome of the project is a tele-monitoring and home support system [65].

Nefti et al. propose a multi agent system for monitoring dementia sufferers. Besides classical sensors (such as, temperature sensor and infrared motion

sensors), the system uses specific sensors, such as natural gas and monoxide sensors, smart cup in order to measure regular fluid intakes, flood sensors near sinks, and magnetic contact switches for monitoring doors and windows [70].

Jeet et al. propose a system in which verbal and nonverbal interfaces are used to obtain an intuitive and efficient hands-free control of home appliances [47].

Alesii et al. propose a solution targeted to people affected by the Down Syndrome. The system provides a presence and identification system for domestic safety, a dedicated time management system to help organise and schedule daily actions, and remote monitoring, control, and communication to allow caregivers and educators sending messages and monitoring the user situation [10].

Lind et al. propose a solution targeted to people with severe heart failure, taking into consideration how an heart monitoring system should work in a contest where users are used to heart monitoring but not accustomed to technology [59].

Innovative Platforms for Wearable Technologies.

Current measures related to health and disease are often insensitive, episodic, subjective, and usually not designed to provide meaningful feedback to individuals [19]. Current research in wearable devices and smartphones opens new opportunities in the collection of those data. A great opportunity comes form Apple that in March 2015 announced Research Kit (RK), an open source framework for medical research that enables researchers that develop iOS applications to access relevant data for their studies coming from all the people that use RK-based applications. Moreover, information will be available with more regularity as people use and interact with their devices. In the following, some example of applications and studies based on RK will be provided.

mPower. The mPower is an app is a clinical observational study about Parkinson disease conducted through an app interface. The app collect information through surveys and frequent sensor-based recordings from participants with and without Parkinson disease. The ultimate goal is to exploit these real-world data toward the quantification of the ebbs-and-flows of Parkinson symptoms [19].

Autism & Beyond. Autism & Beyond aims to test new video technology able to analyse child's emotion and behaviour. The app shows four short video clips while using the front facing camera to record the child's reactions to the videos, which are designed to make him/her smile, laugh, and be surprised. After the acquisition, the analysis module marks key landmarks on the child's face and assesses him/her emotional responses. The goal is not to provide at-home diagnosis, but to see whether this approach works well enough to gather useful data [35].

EpiWatch. EpiWatch helps users to manage their epilepsy by tracking the seizures and possible triggers, medications, and side effects. Data are collected

from sensors and from surveys that investigate the activities performed and the user's state before and after the attacks, and notes about medical adherence [1].

Cardiogram. Cardiogram applies deep learning techniques to cardiology in order to detect anomalous patterns of heart-rate variability, and to study atrial fibrillation, which is the most common heart arrhythmia. Data is collected from people suffering from heart diseases as well from normal one using an app on the Apple Watch [46].

1.5 Architectures for AAL Systems

Section 1.4 gave an overview of some of the available solutions and AAL platforms. This section will provide an overview of the architectural choices of the actual AAL systems. The main aim of this section is therefore to overview the different technological infrastructures adopted for the AAL domain, specifically which are the architectures, the methodologies, the techniques, and technologies used in the design, development and implementation phases of available AAL solutions.

The first issue on the task is that a great number of papers do not report nor seem to adopt any kind of architectural pattern or methodology for the design of the AAL services.

To contextualise such lack of information, [21] proposes a systematic survey of the AAL systems and states that, on a sample of 236 papers presenting various AAL solutions, the majority of them did not report to have followed any particular methodology (157 papers). Among the ones that described the methodology 67% are Goal-Oriented (53 proposals), while the remaining articles are distributed among Agent-Oriented methodology (9), Feature-Oriented methodology(6), and Service-Oriented Methodology (7) as pictured in Figure 1.2.

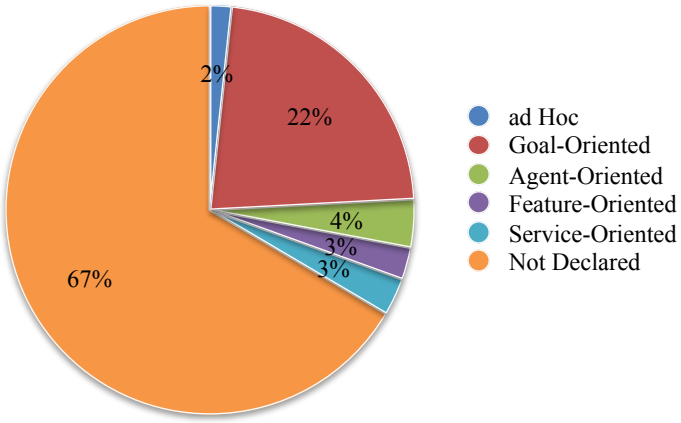


Figure 1.2: Methodologies adopted to design AAL platforms.

Regarding the architectural patterns used, almost half of the papers adopted ad Hoc solutions (51%), while the rest is distributed among Multi-Agent Systems (MAS) (19%), Service Oriented Architectures (SOA) (12%), Client/Server (C/S) structures (8%), Event Driven Architectures (EDA) (2%), and Model Driven Architectures (MDA) (1%). The chart in Figure 1.3 summarises the distribution of architectural patterns in AAL systems.

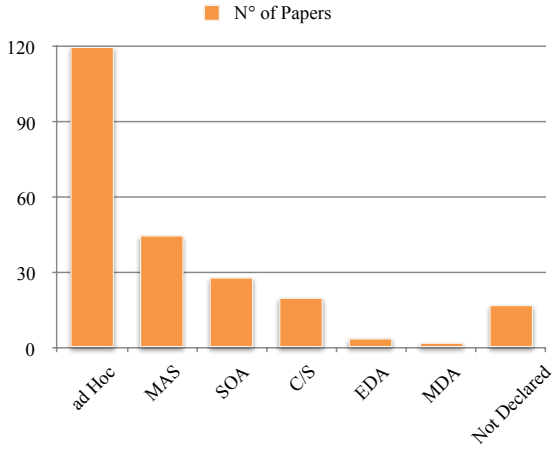


Figure 1.3: Architectures distribution in AAL solutions.

The high distribution of ad Hoc solutions, hints at lack of attention at those quality attributes related to architectural design of software systems. Specifically, these kind of approaches usually result in systems featuring high vertical integration, with tight control of the entire software stack from the end user application to the sensor hardware. The main consequence is a low level of modularity, which is known to be related to low resilience in maintainability low to none ability to reuse some parts of the software.

Moreover, while some available solutions are built on modular frameworks and address the issue of maintainability [13], the other issues mentioned in this work are still to be addressed. As an example the problem of custom configuration of physical sensors, based on each user characteristics has not been covered, as well as the need of low level knowledge at higher levels in order to fully understand the sensed data.

Finally, the usefulness of a standardised way to acquire, represent, and handle data, is confirmed by the fact that the great majority of the identified main activities handled by AAL systems to fulfil their goals are related to inference and manipulation of sensor data[21]:

- Activity Recognition: the identification of Activities of Daily Living.
- Control Vital Status: the monitoring of the vital parameters of patients.
- Position Tracking: the finding or tracking of patients position, both indoor and outdoor.

- Interaction: the activities aimed at allowing the user to deal with assistive technologies.
- Multimedia Analysis: the activities focused on elaboration of multimedia data.
- Data Analysis: the analyses for the discovery of relations, properties, and general knowledge in different kinds of data.
- Data Sharing: the sharing of information and knowledge among the AAL stakeholders.
- Communication: the activities aimed at simplifying the collaboration between users

The different activities and their distribution among the reviewed papers are listed in the chart in Figure 1.4

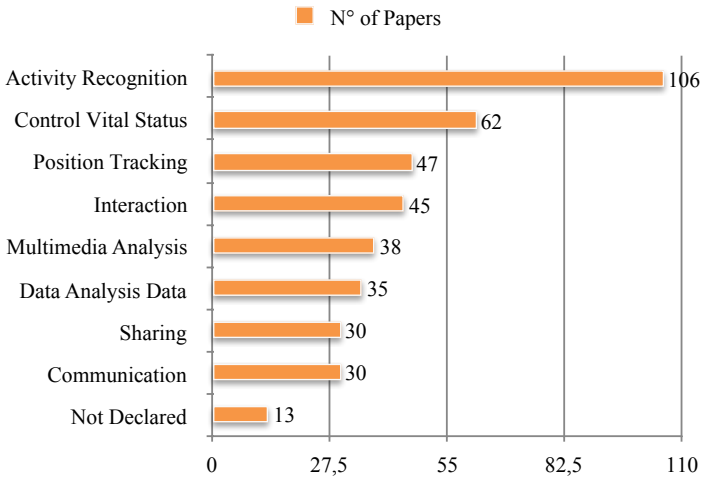


Figure 1.4: AAL-related activities handled in the reviewed papers.

From these analyses arise the need for standard horizontal solutions able to abstract from the specific sensors and data exploited. Such solutions could ease the higher application levels development and adaptation to new sensors and data.

CHAPTER 2

TANA - Timed Acquisition and Normalisation Architecture

This chapter gives an overview of the proposed solution. As mentioned in Section 1.5 most of the AAL systems require acquisition and handling of sensor data in order to fulfil their domain goals.

A well structured architecture should dominate the complexity of such kind of system. The complexity is due, besides to applicative related issues, also to the heterogeneity of the sensors available and simultaneously used. Such architecture should separate the aspects related to the data acquisition, its normalisation, and its fruition. This kind of conceptual separation leads to the realisation of modular systems, whose components are easily changeable, maintained and reused in different contexts. The focus of this thesis is on the *acquisition* and *distribution* of data.

Acquisition The acquisition task requires a set of architectural abstractions that:

1. Simplifies the realisation of customisable data acquisition systems.
2. Controls and expose the acquisition rates at application level without exposing the underlaying complexity.

The former point enables to model and develop large scale systems that can be easily tailored for the different contexts in which they are employed. The latter instead allows to face unexpected situations (such as emergencies) or to adapt to a change of context. As an example if a workout is detected the heart rate sampling rate should increase to better represent the current situation, as well as it should increase in case of supposed heart failure or arrhythmia and decrease in situations where the pulse is known to be lower (i.e. sleep).

Distribution Once the data has been acquired, it should be represented in an homogeneous form regardless of the kind of the sensed information. This approach allow the domain applications to seamlessly integrate and interpret data from new or different sensors without having to learn how such data is structured and how should be interpreted. The data representation should also include aspects related to the timing and to the physical position of the sensed information. A spatio-temporal contextualisation simplifies the realisation of complex behaviours and inferences.

The proposal of this thesis follow this conceptual separation and presents different models for the acquisition activity and the distribution activity. Specifically, the distribution activity will from now on be referred as normalisation activity, as the main focus of the proposed model is on the contextualisation of sensed data in a normalised manner, thus enabling the definition of standardised ways to abstract such data.

The resulting architecture has been named Timed Acquisition and Normalisation Architecture (TANA). TANA covers the activities of acquisition and normalisation of data, and both their logical structures and actual implementations are explained.

The remainder of the chapter is organised as follows: Section 2.1 will introduce the general TANA structure, while Section 2.2 will cover the case studies used for the validation of each activity.

2.1 TANA Overview

As mentioned in Section 1.5, common available solutions and commercial products usually feature a vertical integration of the data pipeline, dealing with all the aspects from the physical acquisition to the domain application, thus reducing flexibility, reuse, and the other issues mentioned in the Section 1.3.

The main goal of TANA is to enable applications to reason on domain specific issues disregarding information about the physical nature and positioning of the sources of data, while instead focusing on meaningful information contextualised in space and time.

In order to achieve such goal, three main activities related to sensor data have been identified: *Acquisition*, *Normalisation*, and *Fruition*.

Data acquisition and normalisation activities have been already introduced, while fruition of data consists in the specific domain applications that ultimately make use of the produced data in order to offer specific services to the users. The fruition of data is therefore very specific to the application domain and thus not part of the main focus of this work. Some specific examples of applications based on sensor data are nonetheless modelled and mentioned as case studies used to verify the feasibility of proposed models. Figure 2.1 pictures the vertical structure of TANA, with its two components at the bottom and domain applications as fruition activities at the top.

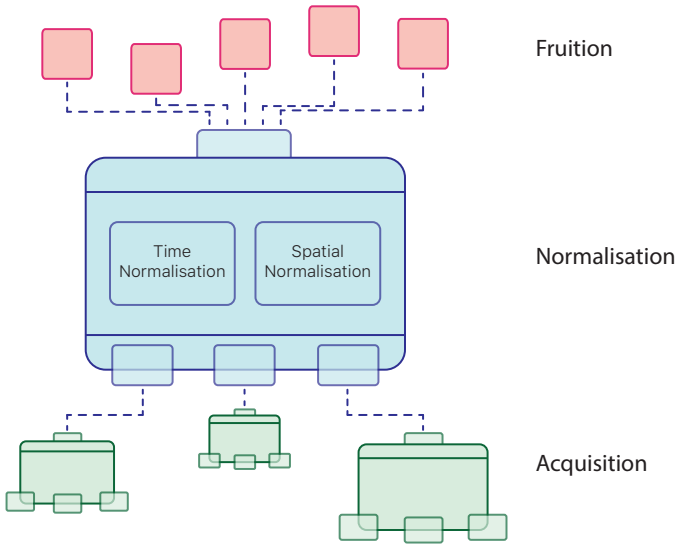


Figure 2.1: Overview of the proposed model.

2.1.1 Acquistion

The activity of acquisition is reified by a set of architectural abstractions named Time Driven Sensor Hub (TDSH). TDSH underlying idea is that time should be a full-fledged first class object that applicative layers can observe and control. Such set of architectural abstractions is supported by a running machine according to the principle of separation of concerns, two different concepts are treated: time drivenness and time observability. A time driven activity is activated by events that are assumed to model the flow of time, while a time observer activity is aware of the current time (in terms of system time). This activities are reified by performers and clocks. Performers are the entities that accomplish time driven activities such as the acquisition of data from physical sensors, while clocks keeps track of the passing time. The consistency of timing for performers activations and clocks updates is performed by the running machine supported by a hierarchically organised set of timers. As an example, an application that exploits heart pulse information, may use TDSH in order obtain a precise timing in data acquisition. In case of a change of context (i.e. emergencies, workouts, sleep) it could slow down all the acquisition related activities, while maintaining consistency among the data.

2.1.2 Normalisation

The Normalisation activity is reified by a set of architectural abstractions defined Subjective sPaces Architecture for Contextualising hEterogeneous Sources (SPACES). The underlaying idea of SPACES is that applicative layers should not deal with intrinsics characteristics of acquisition devices, but should focus

on the provided data, being able to understand and contextualise them without such information about sensors. Moreover the modification or introduction of new sources for similar information should be completely transparent to the existing applications.

SPACES represents a set of architectural abstractions aimed at representing sensors measurements. Such abstractions rely on the concept of *space*. Sensed information is contextualised in time (the time of acquisition) and space, in terms of physical position. A key concept is that in most cases, a physical position of the acquired data cannot be determined by domain applications without information about the originating source and its own position. A clear example is any information extracted from a camera image. In common approaches, the final application can infer the physical position about such data only by directly knowing not only the position of the source of the image, but it also needs knowledge about its inner characteristics (i.e. aperture and depth of field). A well structured data representation should allow to embed positioning information within the data itself, thus freeing the application from the burden of knowledge of every physical devices it exploits.

However, different applications may reason on different spatial models. For this reason the concept of a mapping function, able to take a spatial information related to a specific space and convert it to a different space has been introduced. This allow to have the same information contextualised at different abstraction levels. As an example, any information that has a positioning information in room coordinates, may be mapped to its corresponding values in building or world coordinates, thus allowing each application to access it with the most appropriate spatial information.

Moreover, a similar approach may be applied to the representation of the data itself: at any abstraction level a standard way of representing a kind of data is defined (a *Measurement Space*). Any application that reason at a specific level needs only to know how to interpret data defined in such manner, without having to know how each physical devices originated it.

As with positioning information, by defining this measurements standards at room or building levels, any kind of information that is well contextualised is directly usable by the applications.

Conversion functions act as mapping between different spaces with different data representation. As an example is possible to define a room in which every temperature value must be in Fahrenheit degrees. If at building level, temperature readings are to be handled in Celsius degrees, the conversion function between the two scales acts as a mapping function.

2.1.3 Putting Together

The final result is a set of well defined specifications to represent sensor readings and their positions, having applications only to know such representation, regardless of the underlaying hardware components and any eventual change among those.

Acquisition components obtained following the TDSH principles are meant to be deployed in small, embedded, systems; each one handling a handful of different sensors and providing information to the normalisation level, represented by SPACES enabled components. Each SPACES entity should be able to handle multiple TDSH modules and provide the normalised data to different fruition activities, which may be in different application domains, such as healthcare, home automation, and surveillance.

2.2 Case Studies

In this section, the case studies for the activities of acquisition and normalisation of data are described.

As mentioned, the proposed architecture only covers the two lower activities of an AAL System (acquisition and normalisation); however in order to be aware of peculiarities of the chosen domain, a thorough analysis of fall detection systems have been performed. Moreover a number experiments regarding the various fall detection techniques have been performed, thus covering also the fruition level of the architecture. The results of the fall detection study are published in [63]. Given the familiarity with the specific kind of application, the application scenario chosen as a test case in this work is also about fall detection.

As most of the fall detectors proposed in the literature are based on accelerometers, a similar approach was chosen for this scenario.

2.2.1 The Acquisition Case Study

The main research question handled in the study of an example scenario for data acquisition is about validation: is the TDSH approach feasible? Is it useful? Does it allow the deployment of meaningful application components based on it?

The case study is quite straightforward: a wearable node equipped with a triaxial accelerometer is deployed; accelerometric data can be used in order to perform fall detection, the actual algorithm is irrelevant, as it represents the logic of the domain application.

Extracting the accelerometric data using the principles of TDSH enables to prove the feasibility of the approach. However, the acquisition of a single sensor may be an oversimplification of real application. As one of the most common issues with accelerometer based fall detection systems is an high number of `False Positive` (FP) [56], it is reasonable to assume that an advanced fall detector could feature a second kind of sensor with the aim of lowering the FP. To this end, a variety of different approaches have been proposed in the literature, for the purpose of this scenario an approach similar to [76] has been considered. In addition to the accelerometer, a microphonic array is also deployed. A microphonic array is composed by a set of microphones that must be activated and sampled individually, but consistently with each other. Once obtained, acoustic

information from the array can be used to estimate the height of the source of a specific sound. A loud sound that is originated near the floor is more likely to be related to the fall of a user or an object than a sound that has is origin in mid air.

For the height estimation a number of different approaches are presented in the literature, with the more commons being the Time Difference of Arrival (TDOA) and the Phase Delay. Time difference of arrival is a basic way to obtain a direction estimation of the source and its working principle is based on the different time that takes the same audio waves to reach different microphones at placed at fixed known positions. On the other hand, phase delay is a more complex way to achieve the same goal, it does not imply a strong correlation on every sound sensed by both microphones; usually is achieved by exploiting the Fast Fourier Transform (FFT). In this case, cross correlation between the microphones information must be calculated.

In the sample scenario the final behaviour is then as follows: the accelerometer and the microphones are sampled in a synchronised manner. If the application detects a possible fall by analysing the accelerometric information, it cross checks the result with the microphonic data from the same time window. In case a loud sound has been sensed with a near floor altitude at the same time of the accelerations that originated the first fall-alarm, the fall detector confirms the fall and acts accordingly. On the other hand, if the audio information does not confirm the possibility of a fall, the application may discard the fall-alarm related to the accelerometer as a FP or lower its confidence in the inference.

The addition of a microphonic array has been considered as it represents a logical and state of the art addition to a basic accelerometer-based fall detector. Moreover, starting the validation from the accelerometer and then add the microphones, allows to provide an incremental scenario for the acquisition framework. In particular the acquisition of acceleration data represents a basic setup, with a single sensor. On the other hand, sampling a microphonic array represents a test for a multi-sensor scenario with strict constraints in terms of time-synchronisation (all the microphones in the array must be synchronised with each other).

2.2.2 The Normalisation Case Study

For what concerns the SPACES approach, the main question is about the feasibility of the data representation: can data be represented as proposed in the SPACES model? Does the SPACES approach remove the need of hardware knowledge to exploit the positioning information of the data? Is it flexible enough to adapt to heterogenous sensor data?

The example scenario for SPACES builds up onto the one presented for acquiring data. A first assumption is that the microphonic array is placed stationary within a room that has a spatial representation in the system. As a result, every information acquired from the array, should have a positioning information that relates to such room. This positioning information is not the position of the microphones, but it should represent an estimation of the posi-

tion of the generated sounds in room coordinates; such coordinates have to be obtained from the position of the array and some of its physical characteristics, such as the distance between the microphones. A second assumption regards the position of the accelerometer: it is feasible to assume that a different system component is able to provide the position of the accelerometer within the room. with the accelerometer position available in room coordinates, it is possible to contextualise the accelerations with respect of the room space. It is important to note that in this specific case, the standard contextualisation of the accelerometric data is not trivial: accelerations have directional components; in order to decouple them from the physical device, their values must be calculated exploiting the orientation of the device with respect of the room. Considering a stationary device that only senses the gravitational acceleration as an example, the direction of the acceleration is strictly dependent on the orientation of the device with respect of the room.

As a result, the fall detection application, instead of having to know the physical sensors, may monitor only the information that are contextualised inside the room. Since every information related to the room is well defined and structured, the application should be able to understand every information without dealing with the intrinsic characteristics of the devices.

CHAPTER 3

Time Driven Sensor Hub

In this chapter the design and modelling of the TDSH component are discussed. As introduced TDSH is a set of architectural abstractions aimed at providing consistent timing for sensor acquisition and exposing time-related concepts at application level.

The chapter is organised as follow: Section 3.1 introduces the time awareness principles that represents the foundations of TDSH, Section 3.2 and Section 3.3 give an introduction on microcontrollers and some hints on how the starting model has been adapted in order to be suitable for microcontrollers, Section 3.4 introduces the concrete model of TDSH and Section 3.5 highlights some of the implementation aspects of the actual framework. Finally Section 3.6 covers how the applicative scenario introduced in Section 2.2.1 can be implemented using the TDSH model.

3.1 Time Awareness Machine

Time Awareness Machine (TAM) is a set of architectural abstractions, firstly introduced in [37]. There, the concept of *Time Aware System* is intended as a collection of different *Time Aware Components*, which are in charge of reify activities related to the concept of time: three kinds of time aware activities have been identified:

1. a *Time Driven* activity is triggered by events that are assumed to model the flow of time (for example, it periodically samples incoming data).

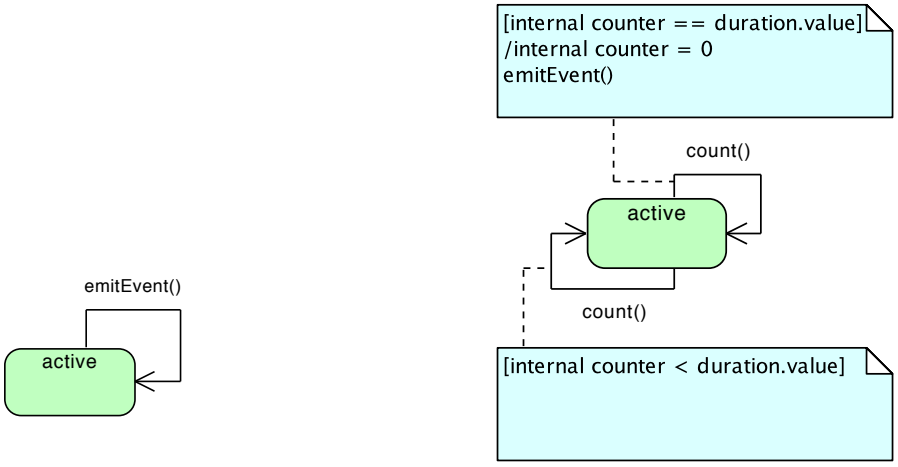
2. a *Time Observer* activity observes "what time it is" (for example, it observes the current time to timestamp the generated data).

3. a *Time Conscious* activity reasons about facts placed in a temporal context, no matter when the computation is realised (for example, it performs off-line statistics on historical data).

The properties of drivenness, consciousness and observability can be enabled by means of three well distinguished concepts (architectural abstractions): Timer, Clock, and Timeline.

3.1.1 Timer

A *Timer* is defined as a cyclic source of events all of the same type: two successive events define a *duration*. A timer generates events by means of its *emitEvent* operation, as shown in the state diagram of Figure 3.1a



(a) Timer (b) Virtual Timer

Figure 3.1: Timer Behaviour.

A *Virtual Timer* is a timer whose event generation is constrained by the behaviour of its *reference* timer: it counts (by means of the *count* operation) the number of events it receives from its reference timer and generates an event when this number equals a predefined *value*. The duration is thus specialised to a *virtual duration*. This behaviour is shown in the state diagram of Figure 3.1b. A virtual timer is also characterised by the possibility to modify the value of its duration (by means of the *setDuration* operation), modifying the speed at which events are generated.

Timers can be arranged in hierarchies, in which every descendant timer has exactly one reference timer. The root of every hierarchy is a *Ground Timer*, which is a timer whose durations are not constrained by the durations of another timer.

Therefore, the durations of a ground timer can be interpreted as intervals of the real external time, so that the events generated by a ground timer can be interpreted as marking the advance of time. Figure 3.2 sketches the described concepts.

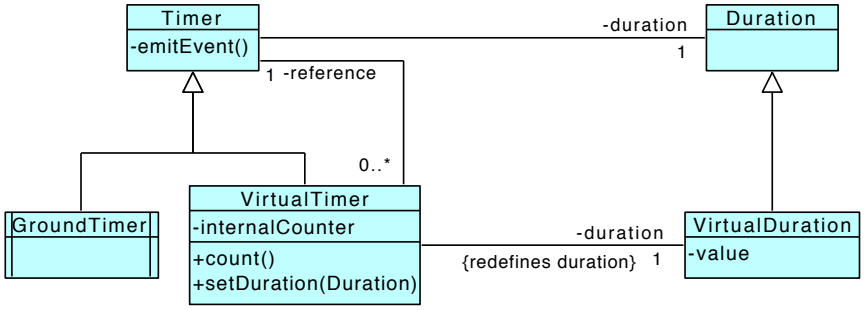


Figure 3.2: Basic Concepts related to Timers.

Using the concepts shown in Figure 3.2, it is also possible to define timers with variable durations, by calling the *setDuration* operation every time a different value is required for the duration. In order to simplify the management of timers with variable durations, it is possible to extend the basic concepts as sketched in Figure 3.3a.

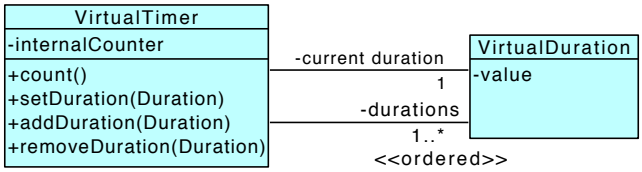
A virtual timer maintains an ordered list of durations that can be modified by means of the *addDuration* and *removeDuration* operations. Figure 3.3b shows the state diagram of a timer with three predefined durations: each time the internal counter equals the value of the current duration, the timer switches to the next duration in the list by means of the *setDuration* operation and then emits the event.

3.1.2 Clock

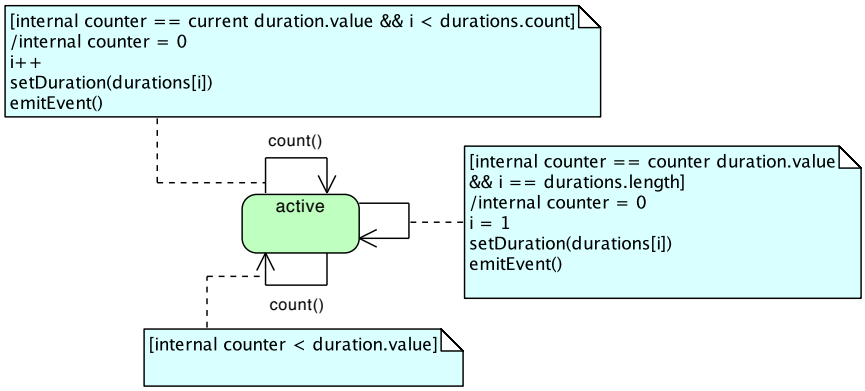
A *Clock* is associated to a timer and counts (by means of its *increment* operation) the events it receives from its timer. The event count can be interpreted as the current time of the clock (see Figure 3.4). Thus, time is not a primitive concept but it is built from events.

3.1.3 Timeline

A *Timeline* is a data structure (thus intrinsically discrete) which constitutes a static representation of time as a numbered sequence of *grains*. A grain is an elementary unit of time identified by its *index* and whose interior cannot be inspected. A *Time Interval*, defined on a timeline, is a subset of contiguous grains belonging to that timeline.



(a) Concepts for timers with variable durations.



(b) State Diagram

Figure 3.3: Virtual Timers with variable Durations.

Figure 3.4: Concepts related to Clocks.

A *virtual timeline* is a timeline whose grains (*virtual grains*) have a *duration* that can be expressed as a time interval in the associated *reference* timeline. Timelines can thus be arranged in hierarchies. The root of every hierarchy is a *Ground Timeline*, which is a timeline whose grain durations are not constrained by the grains of another timeline.

In each hierarchy, the ground timeline is therefore the only one whose grains can be interpreted as an elementary time interval in an arbitrary ground reference time (e.g., the "real" time from the application viewpoint).

With the concept of *Timed* are intended any assertions regarding the system domain that is contextualised in time by a time interval that represents its interval of validity. The concepts related to timelines are shown in Figure 3.5.

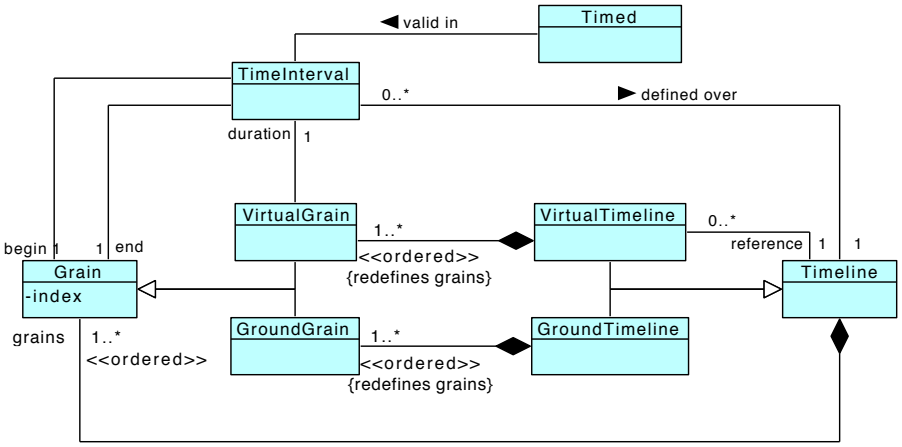


Figure 3.5: Concepts related to Timelines.

By connecting a clock with a timeline, it is possible to interpret as *present time* on the associated timeline the grain whose index equals the clock's current time (see Figure 3.6). Every time the clock receives an event from the connected timer, it advances the present time on the corresponding timeline by one grain.

The clock also defines the concepts of *past* and *future* in the associated timeline: the grains with *index* less than *current time* belong to the past of the timeline and the grains with *index* greater than *current time* belong to the future of the timeline.

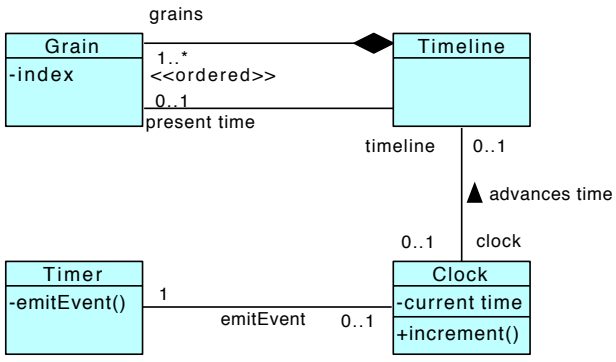


Figure 3.6: Connection of a Clock to a Timeline.

3.1.4 Time Aware Entities

With the term *time aware entity* we denote any kind of entity that performs (by means of the *perform* operation) time aware activities (see Figure 3.7).

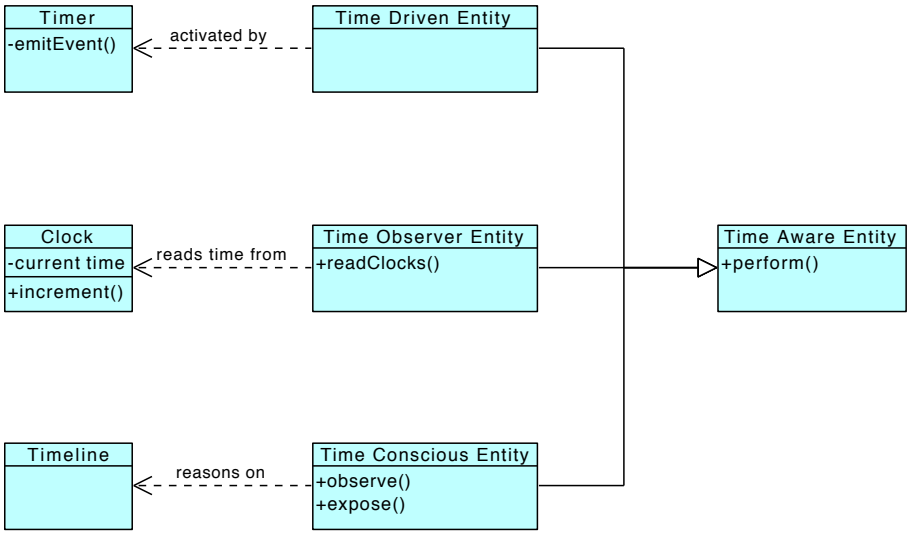


Figure 3.7: Entity classification according to the relation with the basic concepts.

According to the three concepts introduced in the previous sections, the following basic classes of time aware activities can be identified:

- *Time Driven Entity*: an entity whose activation is triggered by a virtual timer when the events generated by the ground timer are interpreted as equally spaced with respect to the real external time.
- *Time Observer Entity*: an entity that reads current time from one or more clocks.
- *Time Conscious Entity*: an entity that reads/writes timed facts on a timeline without any reference to when such a management is actually realised.

More articulated behaviours can be obtained by combining the three basic entities, as sketched in Figure 3.8.

With the term time aware entity is denoted any entity that performs (using a *perform* operation) time aware activities. From the three concepts introduced in Section 3.1, three classes of time aware activities have been identified:

Time Driven Entities

A time driven entity is associated to its *activating* timer, as sketched in Figure 3.9a. As depicted in the state diagram of Figure 3.9b, a time driven entity

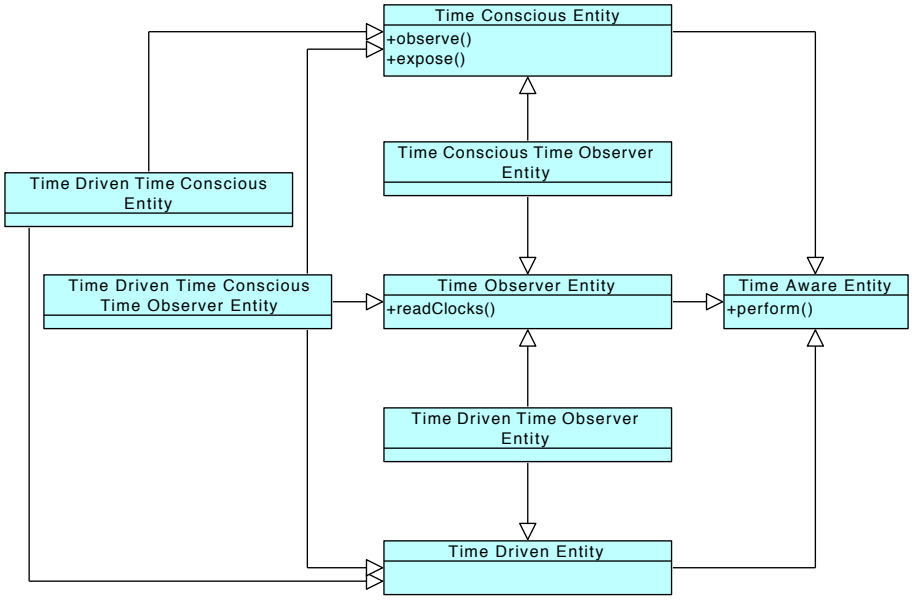
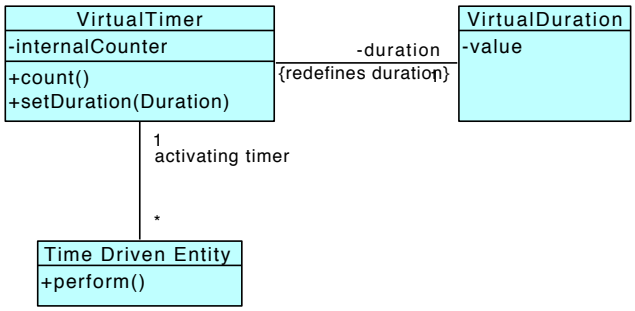
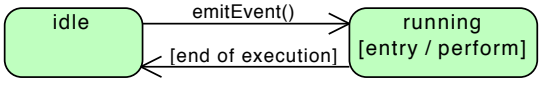


Figure 3.8: Combinations of time-related behaviours.

enters the *running* state when its activating timer emits an event. In this state, it performs its domain-dependant operation. At the end of the execution, the entity goes back to the *idle* state.



(a) Concepts for simple time driven activation.

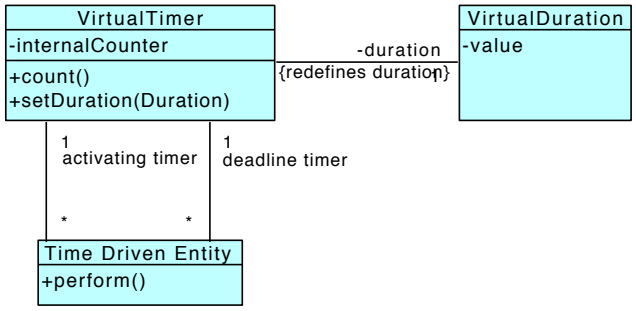


(b) Simplified state diagram for a pure time driven entity.

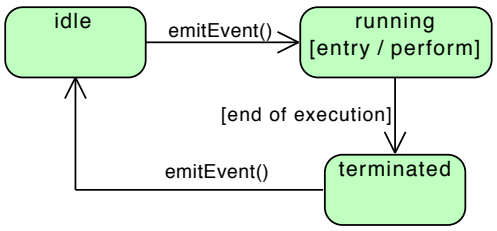
Figure 3.9: Time Driven Entities.

This simple model assumes that the deadline for an execution coincides with the beginning of the next execution. To adapt the model to the general case where deadlines temporally precede the beginning of the next execution, it is possible to associate a second timer to each time driven entity, as sketched in Figure 3.10a.

When the deadline timer emits an event, the associated time driven entity must have already completed the *perform* operation. It follows that a time driven entity must include an additional state, denoted *terminated*, as depicted in Figure 3.10b.



(a) Concepts for advanced time driven behaviour.



(b) State diagram for a pure time driven entity.

Figure 3.10: Time Driven Entities with the Deadline concept.

Time Conscious Time Driven Entities

Some care must be used to guarantee consistency when designing entities that are both time driven and time conscious. In fact, it is desirable that the behaviour of all the entities that happen to be triggered simultaneously does not depend on the order in which the executions are actually managed, which may be affected by low-level details such as the number of available cores or the particular scheduling algorithm that is being used.

It is therefore necessary to guarantee that all the time conscious entities that are triggered simultaneously share the same view of the timelines in which they are interested, to avoid the situation of an entity that reads timed facts written

by another entity triggered simultaneously just because the latter was granted higher execution priority by the low-level scheduler.

A possible solution for this consistency problem is that all entities read timed facts immediately when they are activated by the corresponding activating timer and write timed facts only when they receive an event by the corresponding deadline timer, even if the actual execution ends before the deadline.

The structure that realises this mechanism is described by the state diagram of Figure 3.11, that enriches the one of Figure 3.10b by introducing effects in the transitions triggered by timers. More precisely, the effect of an event from the activating timer is the reading of facts by means of the *observe* operation, whereas the effect of an event from the deadline timer is the writing of facts by means of the *expose* operation.

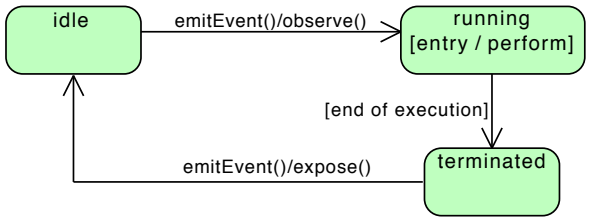


Figure 3.11: Classification of Time Aware Entities and combinations of time-related behaviours.

In an actual implementation, the concrete component in charge of the execution of entities must guarantee that when the execution of a set of entities is triggered, all the entities of the set read timed facts before any one of them is allowed to start the actual execution, and that every entity writes timed facts only at the deadline for its execution.

Moreover, since the interior of a grain is not inspectable, if the timeline is connected to a clock, facts must be written on a timeline only at the end of the present grain. This is realised by means of the *writeTimeds* operation of a timeline. Such behaviour is described in Figure 3.12.

Performers

Performers are entities that accomplish domain dependant time sensitive activities, thus they are time sensitive entities. It follows that their activation is triggered by the ticks of a timer. The performer activity is reified by the *perform* operation, whose duration must be less than or equal to the period of the ticking timer to fulfil real-time constraints.

As depicted in Figure 3.13a, two states characterise a performer: *waiting* and *running*. When its ticking timer ticks, the performer passes in the *running* state, which implies the execution of its *perform* operation. When this operation is completed, the performer passes in the *waiting* state.

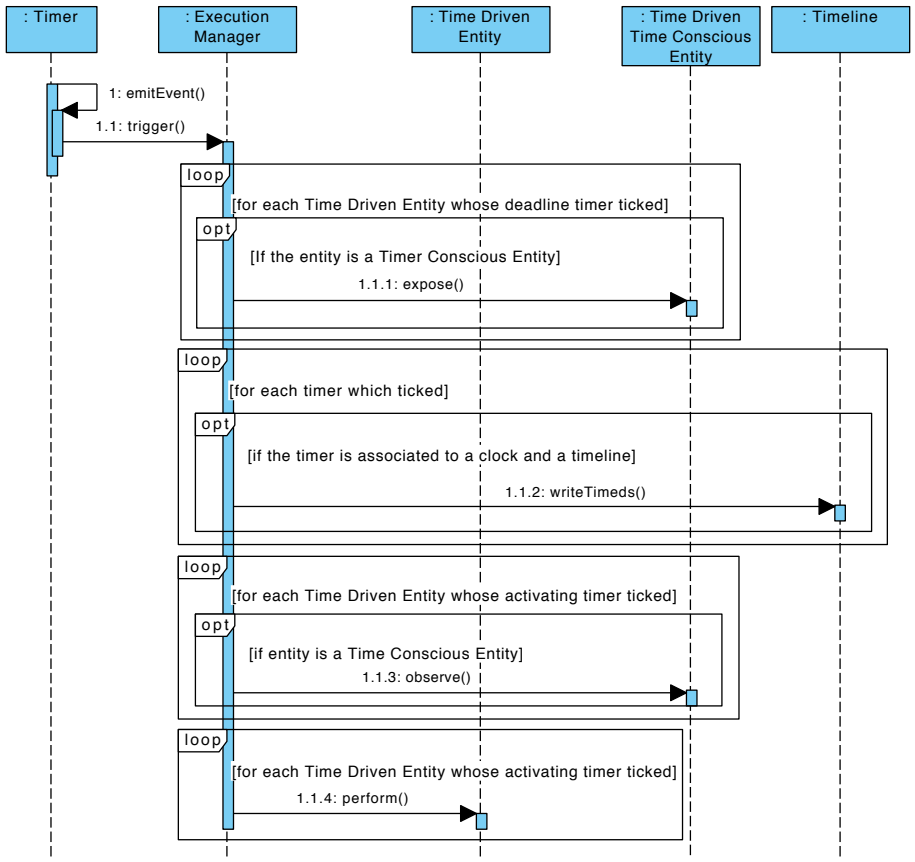


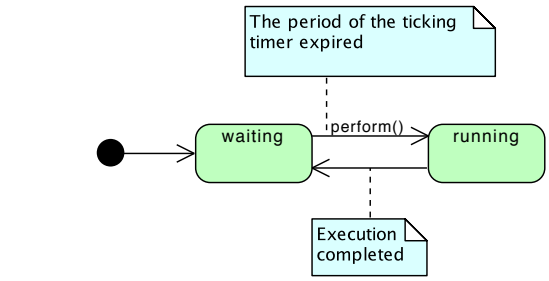
Figure 3.12: Classification of Time Aware Entities and combinations of time-related behaviours.

Performers can be further classified. A *time conscious performer* reads and writes timed data from or to one or more timelines.

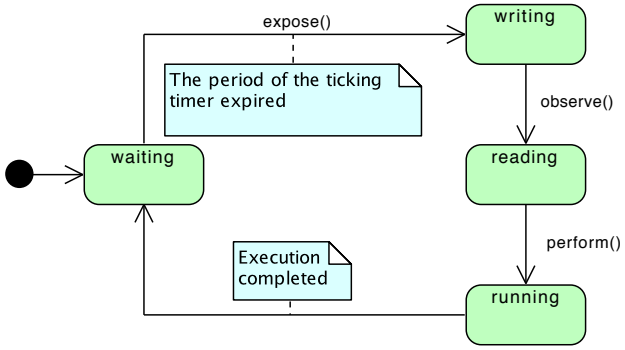
A *time observer performer* reads one or more clocks to get their current times. The states of a time observer performer are the same as the ones of a pure time driven performer.

Different is the case of time conscious performers, since they deal with timelines. A time conscious performer is expected to read timed data, to perform some operations (possibly on the read information) and to write new timed information. However, the duration of the *perform* operation is in general not negligible and can be shorter than the duration of the grain.

Because time is discrete and what happens inside a grain is not observable, timeds cannot be written directly at the end of the *perform* operation. Thus, a performer must read from timelines at the beginning of a grain and can write on them only at the end of the grain. As the end of a grain is the same as the



(a) Performer States.



(b) Time Conscious Performer states

Figure 3.13: States diagrams of performers.

beginning of the next one, writing at the end of a grain is equivalent to writing them at the beginning of the next grain.

Therefore, a time conscious performer writes information at the beginning of the next grain. This behaviour is depicted in Figure 3.13b. The *expose* operation writes timed information to timelines, whereas the *observe* operation reads them from timelines.

3.2 Microcontrollers

A *microcontroller* (or MicroController Unit, also known as MCU) is usually denoted as a device where CPU, memory, and I/O capabilities are wrapped in an integrated circuit programmed to perform a specific task. Usually I/O is performed towards switches, LEDs, sensors, and actuators.

According to Moore's Law microcontrollers have shrunk in size and increased in processing power, allowing to embed logic and computational power even in disposable objects, such as smart boxes for commercial products. Nowadays MCUs are embedded in the majority of appliances, gadgets, and other electronic devices.

3.2.1 Anatomy of a microcontroller

From a technical point of view, the basic internal designs are similar among MCUs. The typical components of a microcontroller are shown in Figure 3.14.

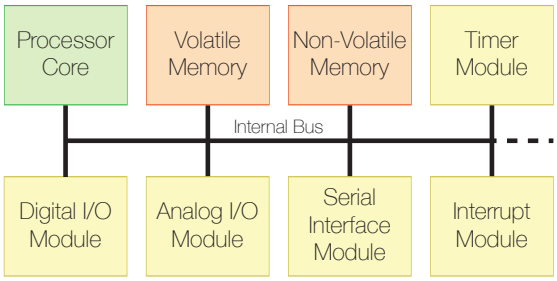


Figure 3.14: Basic components of a microcontroller.

Processor Core: The CPU of the controller. Being a stripped down microprocessor, it contains the arithmetic logic unit, the control unit, and the registers.

Volatile Memory: Is the memory used by the MCU for temporary storage and peripherals configurations. In microcontrollers it is usually SRAM (Static Random Access Memory), which does not require a new electrical charge every few milliseconds like DRAM (Dynamic Random Access Memory) that is the most common kind of RAM found in personal computers.

Non-Volatile Memory: It may be split in program memory and data memory with the former being used to store the programs of the microcontroller and the latter for other data. Memory in this category includes common ROM (Read Only Memory), PROM (Programmable ROM), EPROM (Erasable PROM), EEPROM (Electrical EPROM) and FLASH. Most common and usable solutions uses EEPROM as program memory: it cannot be changed by the program running on the MCU CPU, but it can be erased and rewritten from a personal computer by means of specific programs.

Timer Module: Almost every controller has at least one, and usually two or three timers that can be used to timestamp events, measure intervals, and count events. Many controllers also contain PWM (Pulse Width Modulation) outputs, which is relies on timers for controlling its duty cycle. In critical systems, where microcontrollers are vastly adopted, it is important to be able to detect errors and deadlocks in the program and the hardware itself, for this reason many controllers embed watchdog timers that are used to reset the unit in case of "crashes".

Digital I/O Module: It allows digital/logic communication with the MCU and the external world. The signals are that of TTL (Transistor Transistor Logic) or CMOS (Complementary Metal-Oxide Semiconductor) logic. Digital ports are one of the main features of microcontrollers and their numbers varies from just a few to almost one hundred, depending on the controller family and type.

Analog I/O Module: It is based on ADCs (Analog to Digital Converters) that may feature from 2 to 16 channels and usually has a resolution between 8 and 12 bits). It often includes DACs (Digital to Analog Converters and analog comparators.

Interrupt Module: Interrupts are used for interrupting the program flow in case of events, both external or internal, deemed as important. Basically they enable the microcontroller to monitor certain events in the background, while running its program and react to such events by temporary pausing the main program to handle the interrupt.

Serial Interface Module: Controllers generally have at least one serial interface that is used to download the program and for data communication in general. An example is the USART (Universal Synchronous/Asynchronous Receiver/Transmitter) peripheral that utilises the RS232 Standard. Many controllers also offer others serial interfaces, like SPI (Synchronous Peripheral Interface) and SCI (Serial Connect Interface).

Moreover, many microcontrollers integrate bus controllers for common busses such as IIC and CAN. Larger controllers, may also embed PCI, USB or Ethernet interfaces.

Finally some units are also equipped with additional hardware for debugging purposes, an activity that on microcontrollers is not as straightforward as it is on desktop environments.

3.2.2 Available Microcontroller Boards

As explained, microcontrollers are technically the single silicone chip that includes all the components mentioned in Section 3.2.1.

However, likewise normal CPUs in desktop environments, a single MCU is useful only if a surrounding hardware environment is present. In the case of MCUs they are usually called *Single Microcontroller Boards* (simply board, from now on)[66].

A board is a printed circuit that contains all the necessary elements to exploit the functionalities (ADCs, communication means, etc.) of an MCU. The final goal of boards is to be immediately useful to an application developer, without having to re-design and build the controller hardware. Due to their increasing computational power and communicational capabilities some boards are now

also defined as single board computers. Here some of the most known boards are introduced.

Arduino

Arduino[1] is probably the world most known family of MCU boards. It is an open source computing platform based on low power microcontrollers, as well as a minimalistic development environment. Being open source there are a number of clones, variations and accessories; but a standard Arduino board is equipped with an 8, 16 or 32 bit AVR (a family of RISC microcontrollers) produced by Atmel. Early versions used to be programmed via RS232 ports, but current models are easily programmed by USB and some specific models allow programming over Bluetooth or WIFI connections. The Arduino family is quite basic, with low computational power and and memory, but it stands from the competition for the great community and material available online.

Raspberry Pi

The Raspberry Pi Foundation[2] is a non-profit organisation from the UK and they built the first Pi board in order to bring back the kind of simple programming and tinkering typical of the 1980s. It is built on the Broadcom BCM2835 and BCM2836, which are multimedia application processors geared towards mobile and embedded devices. They include ARM processors in the order of hundreds of MHz, GPU, 512Mb to 1 GB RAM, and MicroSD socket for boot media and persistent storage. The Pi family, opposed to the Arduino, supports complex Operating Systems such as many Linux distributions and (limited to the last model) Windows 10. Its main programming language is Python, but Java, Ruby, and C/C++ are also supported.

BeagleBoard

The BeagleBoard[3] is the response from Texas Instrument to the Pi success. It is open-source, based on ARM architecture and its focused on graphic rendering and output, with some models having HDMI output. The standard models are equipped with 1 GHz ARM processors and feature either wired or wireless network capabilities. An SD card reader and USB are also present. Similarly to the Raspberry Pi, the BeagleBone is more focused on high level programming (supporting a number of desktop level Operating Systems) rather than low level applications that deal with sensor, while still supporting such functionalities through standard 12 bit ADCs.

[1] https://www.arduino.cc
[2] https://www.raspberrypi.org
[3] https://beagleboard.org

STM32

The STM32[4] is a line of products from STMicroelectronics. It has been built for cost-conscious applications that require performance, connectivity, and energy efficiency. They are equipped with different 32 bit RISC ARM processors, such as the Cortex-M0 or the Cortex-M4. It is the third family of single board computers from STMicroelectronics. The F4 series has been the first to mount the Cortex-M4 and to support DSP and floating point instructions.

The Arduino boards, while perfect for simple and quick projects, are generally too simple and limited for supporting complex architectures and frameworks onboard.

On the other hand, boards like the Raspberry Pi and the BeagleBone support reading and reasoning on sensor data, but their focus is more on applications that require visual output and their general usage of high level Operating Systems make them non suited for low level, real-time operations and reasoning.

These are the main reason for which the final choice fell on the STM32 family from STMicroelectronics, specifically on the STM32F4 board. There are of course a great number of other possibilities, but the STM32F4 boards are quite well documented, with a fairly active community and are easily available.

3.2.3 Software Architectures and Embedded Systems

Microcontrollers, both in commercial available boards or ad hoc solutions, are employed in what are called *Embedded Systems*. An embedded system usually relies on microcontrollers, but solutions based on standard microprocessors are also present.

Embedded systems are dedicated to specific tasks, which allow them to be highly optimised in comparison with standard software components on general purpose machines, they are usually considered part of a bigger system (e.g. the control system of a washing machine) and interact with the real world.

Software architecture nowadays represents an area of intense research. The literature offers a great variety of proposals for different application domains that cover both desktop and embedded systems. However, software architectures are usually strictly related to the application for which they are proposed and there is the lack of general approaches that tackle those issues that are peculiar or more critical in embedded systems. [93] is one of the few examples that applies general architectural considerations that are independent from the application domain but common among embedded systems, such as efficiency, hardware limitations, and architecture-compliant implementations.

One of the main questions is deciding which aspects of an embedded software system have to be considered critical from an architectural perspective. It is reasonable to assume that as in standard architectures components, connectors and how they are configured are considered as basic building blocks, but further

[4] http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169

aspects should be taken into consideration. As an example in embedded development, software may be built only based on specifications, with the actual platforms being non-available or even non-existent at time of development.

3.2.4 Software Development for Embedded Systems

Software development for embedded systems is in large part comparable to development for a common workstation, however, there are some crucial differences that make development for embedded systems more complex.

Actually, despite the fact that there are no conceptual differences to the development process, due to the peculiarities and limitations of direct hardware access, the challenges faced by developers are quite different than whose working in high level environments.

High level programming is "just" software development and there is the perception that embedded development is similar, but done at the software side of the border between software and hardware. In fact, embedded development requires to constantly have to "cross" over to the hardware side of the system, in order to fully understand the interaction between the microcontroller software and the associated hardware. It is important to understand what happens to the source code *after* it has been compiled, examples are how the various data types are actually mapped in memory or which optimisations are done by the compiler or seemingly right source code, might not produce the expected results.

On top of such conceptual differences, embedded development presents some physical differences as well: as an example target systems are generally small and dedicated and do not provide any support for software development. Debugging support is usually limited or inexistent, while debugging needs have the additional elements of timing and hardware behaviour. Hardware issue such as spikes on data due to noise are very hard to identify and usually produce intermittent and transient failures.

The fact that debugging support is generally scarce is important as an embedded software system is typically developed and tested in a simulated environment as the target hardware, as mentioned, may be still inexistent or other way unavailable. This fact may represent an issue as the characteristics of the target environment directly affect certain software decisions, such as the distribution of components or the means of communication.

Finally, as mentioned, there is also the issue of resource constraints to consider. While nowadays memory usage and optimisation are often negligible in desktop environments they are key aspects in embedded systems. Power consumption is another example, while it is of no concern in common development it becomes pivotal in battery-powered embedded systems.

Due to all these reasons, development of embedded solutions requires to be particularly meticulous and patient in finding bugs that may be inherent to the hardware itself.

3.3 TAM for Embedded Systems

Time critical components of applications are usually modelled as small dedicated nodes that embeds and handle the strict time constraints, while the rest of the application may reason on more relaxed timings. The TAM framework is aimed at modelling such components, however the current design has been validated only through a desktop based solution (using the Java language). As explained in section 3.2.4 designing and developing architectural abstractions and frameworks for embedded systems requires a dedicated evaluation of characteristics such as computational complexity and memory usage.

In order to better suit smaller target systems, the set of available structures and features have been examined, re-designed and, in some cases, eliminated. As an example, while all the three time-related aspects are treated, time-conscious entities (timelines) had to be redesigned and are explained in 3.3.2. Moreover in order to maintain consistency and reduce the system overhead in terms of memory, only a single clock is considered for each framework instance. This restriction avoid the need to keep track of changes in timers duration that would impact the proportion of the duration of virtual grains with respect to the grains of the ground clock.

3.3.1 Performers and Durations

In Section 3.1.4 the description of performers is given. However due to implicit limitations of an embedded platform (discussed in Section 3.2) only their basic behaviour has been modelled. Moreover, the general lack of multiple cores and no common support for multithreading as hardware and software limitations, impose a more restrictive assumption about performers duration.

Not only the duration of a performer execution must be less than the time that elapses between the performer activations, but it must be lower that the fastest time grain considered in the framework and counted by the ground timer. In fact, the execution of all the performers of the system combined, should be lower than the configured resolution for the platform itself.

The reasons for this strict limitations are related to the sequential execution of a single-thread application on a single-core machine, more details are given in Section 3.5.3, where the actual implementation is discussed.

3.3.2 Timelines, Timeds, and Buffers

The concept of timeline has been introduced in Section 3.1.3. The concrete model of a timeline is however more complex and includes a dedicated manager in charge of handling the storage of timed information put inside its timeline. This allow to have different policies in terms of handling of data.

A basic manager stores the facts in RAM as a List at runtime, but looses any information at shutdown. A more complex manager also stores the information on a persistent storage memory for further access and analysis. Moreover other managers can be easily added to the framework in order to exploit different

patterns for data handling (as an example, a database-based approach has been looked into). Figure 3.15 shows the definition of the timeline in the desktop implementation.

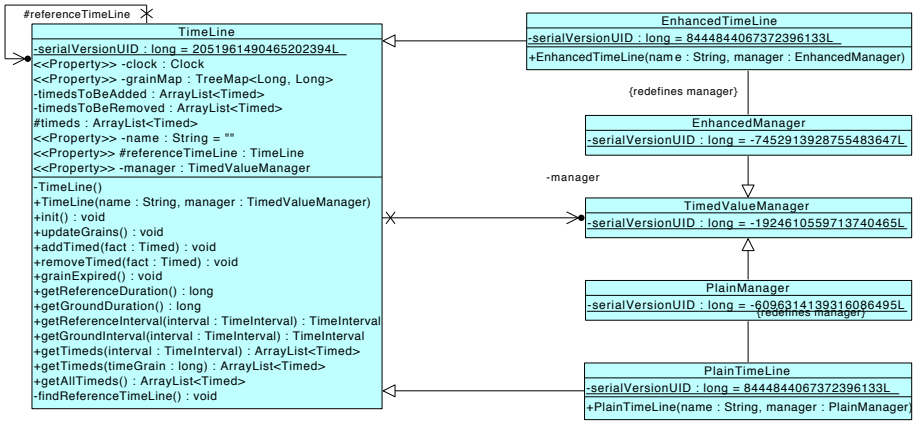


Figure 3.15: Structure of a concrete Timeline.

Such a complex design is too cumbersome for embedded platforms and the concept of timeline had to be revisited. In order to lighten the framework some assumptions had to be made:

- The only purpose of a timeline is to act as blackboard for timed information, in fact enabling the share of such information.
- Long term storage of timed information, if needed, has to be assessed explicitly and outside timelines.
- In order to reduce memory usage and to be able to perform consistent estimations of the memory needed for the systems, their capacity should be limited.
- Components that read and write information into timelines are responsible for correct memory allocation and deallocation of the data they write and read.

From assumptions such those, a simpler model has been chosen, based on a *CircularBuffer*. The model of a concrete buffer is showed in Figure 3.16, where *Data* is the corresponding of a *Timed* as described in Section 3.1.3: it keeps the same characteristics of being a container of any kind of information, as long as it is enriched with a timestamp.

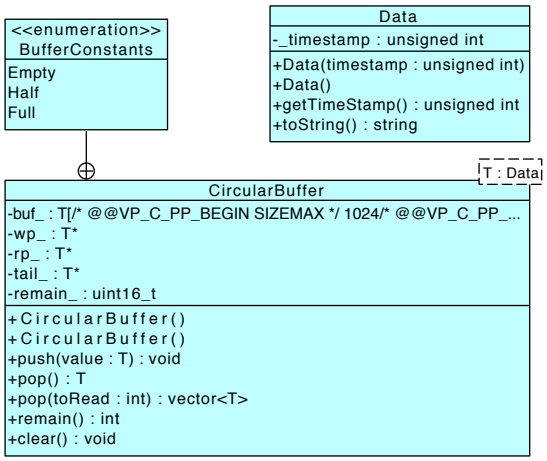


Figure 3.16: Structure of an embedded Timeline.

3.4 TDSH Concrete Architecture

The concrete design of the Time Driven Sensor Hub is based on the main TAM concepts presented in Section 3.1. The focus on embedded platforms and their peculiarities as mentioned in Section 3.2 led to specific design choices such as the ones mentioned in Section 3.3. The main classes of the resulting design are pictured in Figure 3.17.

The `Ticker` is the software representation of the external time trigger and it is considered as the time source for the TDSH framework. The `Timer` entity represents the base of the TAM model: a specific timer (named the GroundTimer) is connected to the ticker and *ticked* by it. Each timer may have a number of sub-timers to which it acts as reference.

Every timer is also characterised by a *period* that determines upon how many ticks of its reference timer it has to trigger. This mechanism act as a *prescaler* and it allows to obtain a hierarchy of timers where the groundTimer is the root of the hierarchy.

Each timer may have a related `Clock` and the clock attached to the groundTimer is named `GroundClock`. As in TAM, a clock counts the time passed since the startup of the systems in terms of ticks of the timer it is associated to.

`Performers` are the reification of purely time-driven entities and represent the operative nodes of the framework. Performers are attached to timers and their execution is triggered by them when their internal counters match their periods.

By chaining timers and performers like discussed in Section 3.1, it is possible to design a system in which operations are strictly timed and it is possible to know exactly *when* any information has been produced in terms of local time, by looking at the clock corresponding to the timer that triggered the performer execution.

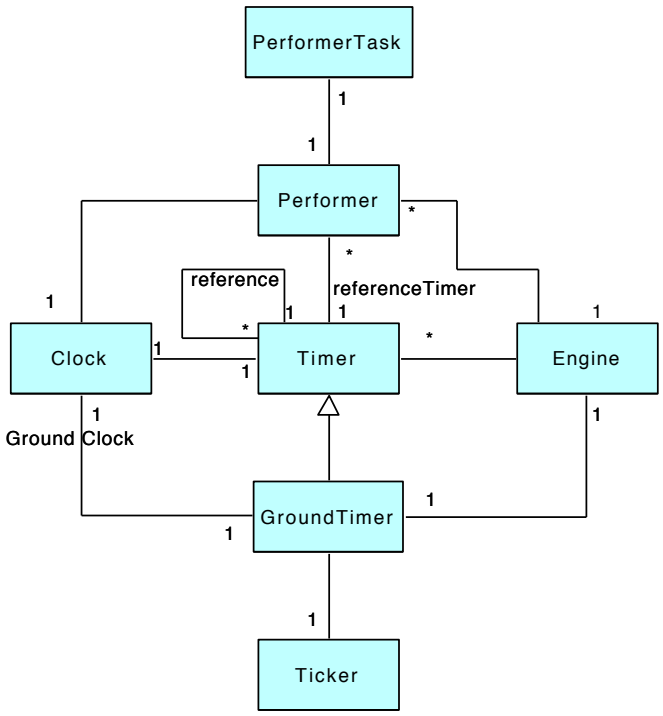


Figure 3.17: TDSH base classes.

The system behaviour is characterised by the following states:

- Built, it is the state in which the system enters after the configuration of the framework topology has been loaded.
- Ready, is the state representing the fully configured system (topology + initial states), but it is not running.
- Running, it represents the state in which the system is running.
- Suspended, it is the state in which the system has been paused, it stops the advance of time from the framework point of view.

The operations enabling the transitions between the states are described in Table 3.1.

3.4.1 Timer

The structure of a Timer, retains most of the characteristics from the original TAM model, nonetheless its behaviour has been better defined: the possible states of a timer are pictured in Figure 3.19:

Table 3.1: Timers states transitions

Name	Parameters	Behaviour
`setConfiguration`	Configuration `configuration`	Loads the topological configuration of the system.
`setState`	SystemStates `initStates`	Configures the states of timers and performers (their periods and current state). To maintain consistency, it cannot be invoked while the system is running.
`start`	-	Starts the system after it has been fully configured.
`pause`	-	Interrupts the execution of the system, without modifying the configuration.
`resume`	-	Recover the normal execution of the system after it has been suspended.
`shutDown`	-	Stops the execution of the system and removes the framework instance from the memory.

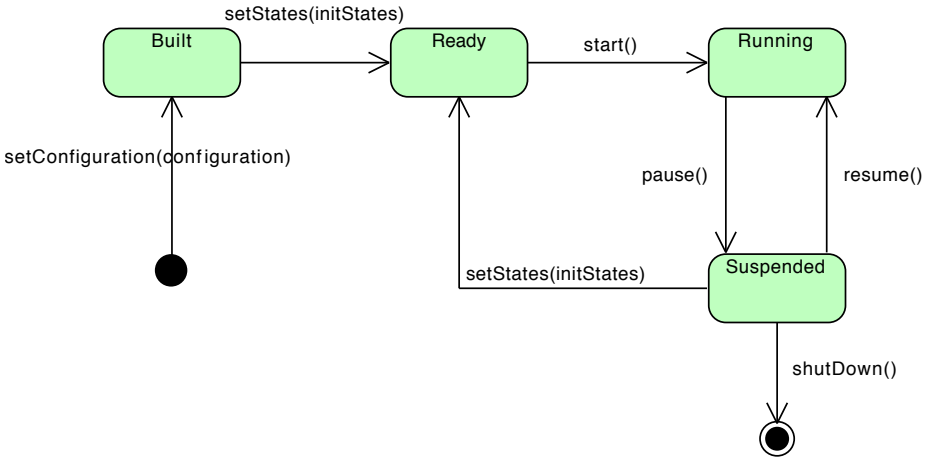


Figure 3.18: States of the System.

- Built, it represents the state in which a timer is after its instantiation.

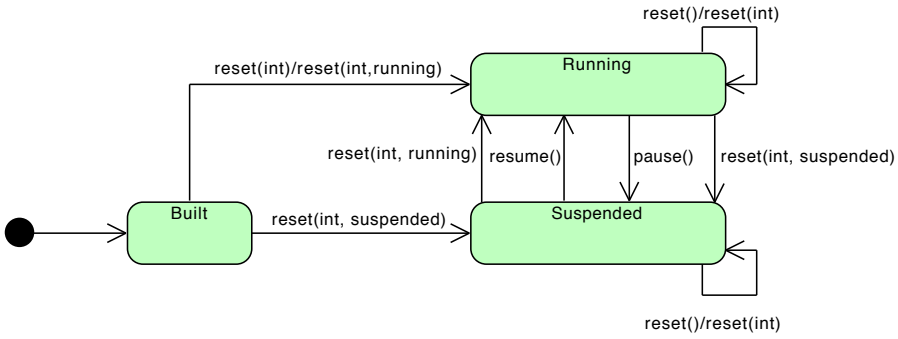


Figure 3.19: States of a Timer.

- Running, its the state of execution. It is the default state for an initialised timer (which is an instantiated timer upon which a valid period value has been set).
- Suspended, it represents the state of a timer that has been put on hold. It is also possible to initialise a timer in this state instead of running.

The transitions between such states are instead described in Table 3.2 and are labelled with the stereotype `<<Runtime>>` in Figure 3.20 that pictures the concrete design of a timer. the operations stereotyped as `<<Configuration>>` and `<<Reflection>>` are, as the names suggest for configuration of the timer and its related elements and for reflection purposes, these aspects are treated in Section 3.4.4.

Figure 3.20 also shows how a Timer holds reference to all of its `subTimers` and `performers`. Moreover it have a `reference` that links it to its reference timer.

The operational behaviour of a timer is pictured in Figure 3.21, when a timer (*t1*) is ticked, it increments its internal counter and if it matches its current period it emits an event. The event emission implies ticking all of its sub-timers (*t2*) and setting its performers (`p`) to be executed. The execution of the performers is not contextual to the timer execution and it is handled by the Engine, as described in Section 3.4.3.

Table 3.2: Timers states transitions

Name	Parameters	Behaviour
`pause`	-	Stops the execution of the timer, which means that it will not increment its internal counter.
`resume`	-	Re-starts the execution of the timer (internal counter increment, tick propagation and performers execution).
`reset`	-	It does not change the current state, but it clears the internal counter.
`reset`	int `newPeriod`	It does not change the current state: it clears the internal counter and sets the period value to `newPeriod`.
`reset`	int `newPeriod` TimerStates `newState`	It clears the internal counter, sets the period value to `newPeriod` and changes state to `newState`.

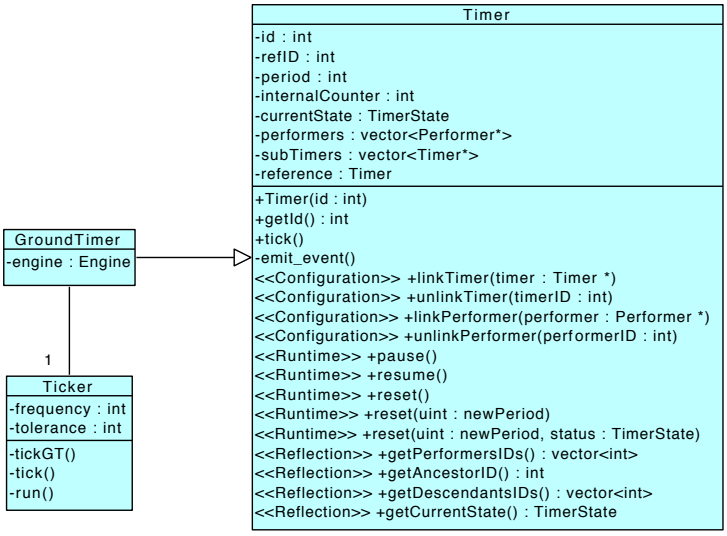


Figure 3.20: Concrete design of a Timer.

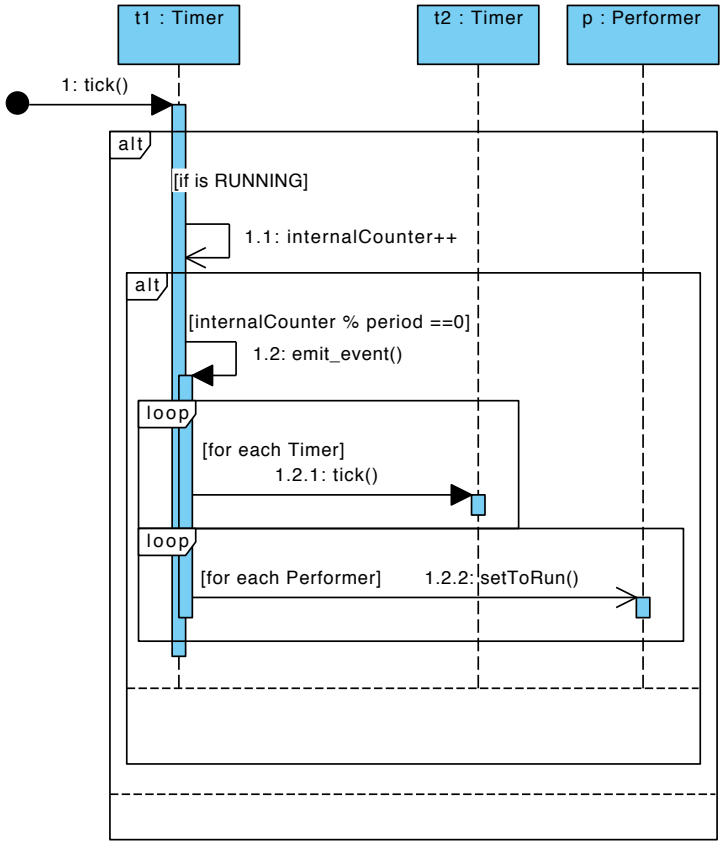


Figure 3.21: Execution of a Timer.

Ground Timer and Ticker

The timer that starts the tick propagation is the `GroundTimer`. Its behaviour is bounded to the global state of the system pictured in Figure 3.18. The ground timer is defined as a timer that holds a reference to the `Engine` and it is referenced by the `Ticker` as pictured in Figure 3.20. The attributes of the ticker are used in order to define the base frequency of the system (in milliseconds) and the tolerance for delays.

As shown in Figure 3.22 whenever the physical ticker interface `ticker` (which is external and independent from the system) ticks the ground timer *gt* it increments the ground clock `gc` and then iteratively ticks all the timers that are directly attached to it. After the clock and all the timers have been ticked it start the execution of the `Engine`, discussed in Section 3.4.3.

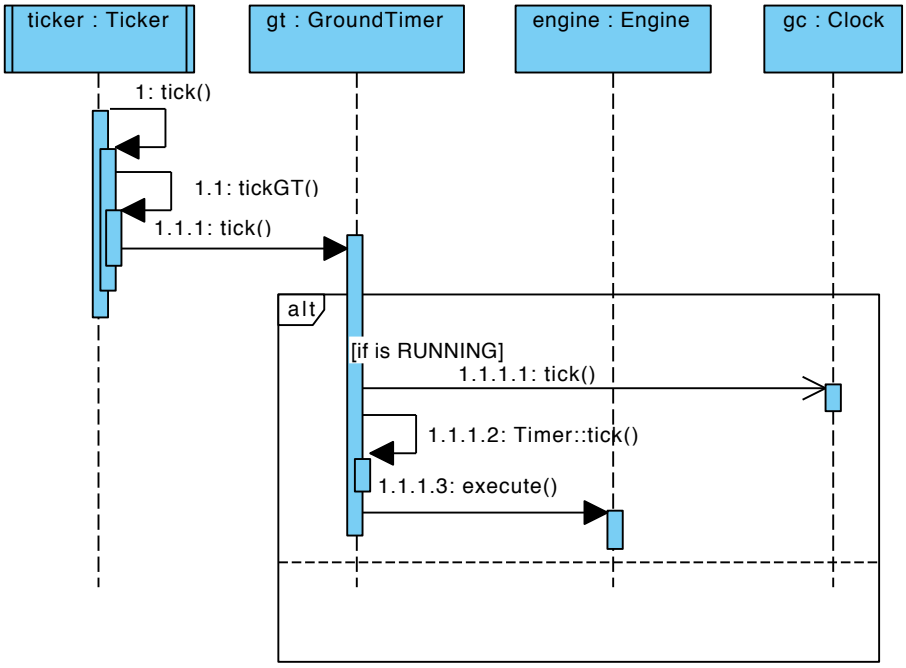


Figure 3.22: Execution of the Ground Timer.

3.4.2 Performer

As pictured in Figure 3.23, the concept of performer has been reified in two different entities: the `Performer` and the `PerformerTask`. The performer reifies all the characteristic of a TAM performer (see Section 3.1.4), with some additional functionalities in order to handle its buffers. The PerformerTask is a simple

entity that has the only duty of defining the task that has to be performed. It can also use the `write*` routines in order to store and diffuse information.

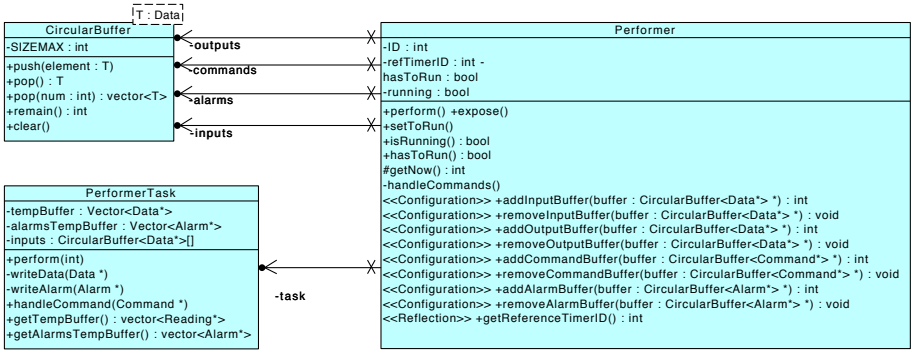


Figure 3.23: Concrete design of a Performer.

While it is easy to see the task being performed (hence, the performerTask entity) as a state machine, the performer itself does not have any particular state definition, since its state is bounded to the one of its activation timer.

Buffers and Data

As mentioned in Section 3.3.2 The concrete design of timelines has to face issues such as memory management and thus a more low level practical solution has been proposed. Instead of generic (illimitate) timelines, there will be a set of `Buffers` that performers can read and write; specifically, the considered structure is that of a `CircularBuffer`.

Circular buffers allow to avoid an uncontrolled growth in memory usage (as they are limited), but are more flexible than normal arrays. Moreover, in case of need older data would be erased in favour of newer timed information, which is by assumption more useful and meaningful since it represents a more recent state of the environment.

CircularBuffer is a template class and it can be instantiated with a generic object type named `Data`, also introduced as the concrete counterpart of the concept of *timed*.

Standard interactions are designed as follows:

- `push(T)` is the function used in order to add a data element to the buffer.
- `pop()` will return the first element of the buffer, which means the older element still on the buffer.
- `pop(int)` is equivalent to the `pop()`, but will return a vector of data elements composed by the last `n` elements of the buffer.
- `remain()` returns the number of items still present on the buffer.

- `clear()` empties the buffer without returning any element.

While the parametrisation of the buffer with the `Data` type allows more flexibility and reuse, TDSH is devoted to sensor based application, which is why more data types have been designed in order to take account of different behaviours.

For example, an application may be interested in sending a specific `Command` to a running performer, like the change of a threshold; on the other hand, the same performer may be interested to produce an `Alarm` if its readings are over such threshold.

This specialisations lead to the definition of the data hierarchy pictured in Figure 3.24.

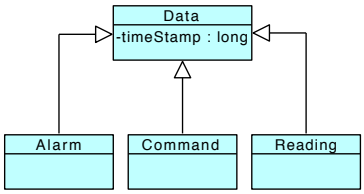


Figure 3.24: Data Hierarchy.

The assumption is that commands may easily be handled differently from other input data and they may have to be dealt as soon as the performer is activated. Similarly alarms may be sent to a dedicated status-controller performer and not as a performer usual output.

As an example, Consider the scenario of a simple thermostat, where there is a temperature probe that is polled periodically: each information is sent to the central station for statistical purposes and, in case the temperature value is below some given threshold t, it turns on the heater. The TDSH setup for this scenario can be the one pictured in Figure 3.25. Each performer is designed as follow:

- P_1 is the acquisition performer, its activation timer VT_1 has an activation period of 6. VT_0 has the same timing of the ground timer GT, which is itself paced at a 10 seconds period, giving P_1 an acquisition rate of 1 per minute. Each value polled from the probe is stored in a `TemperatureReading` value, which is a specialisation of the `Reading` type that also holds a float value for the temperature. Finally, every `TemperatureReading` is placed both in buffer B_1 and B_2.

- P_2 is the temperature monitor, in case the temperature values obtained through P_1 are below the given threshold `t`, it produces an alarm and put it inside its buffer B_3. Also, P_2 features a second buffer, B_4, that is used as input and in which commands for changing the threshold values can be placed.

- P_3 represents the communication module: it collects information from both buffers B_1 and B_3 and then sends them outside the system through, as an example, a serial communication using the bluetooth standard. Since most of today communication standards for embedded platforms features bidirectional communication, it is reasonable to assume that it also handles incoming communications from outside the system such as the commands interpreted by P_2: for this reason it features B_4 as one of its buffers and it writes on it the commands received.

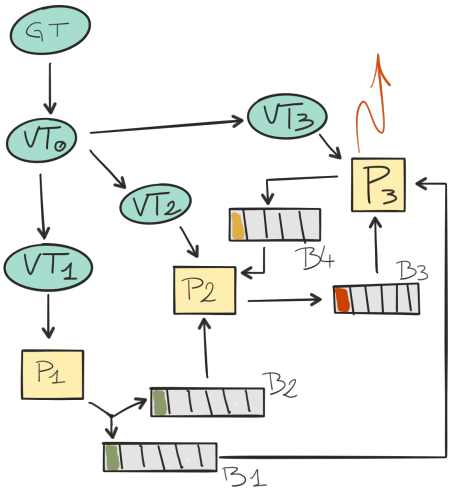


Figure 3.25: Buffer example.

In order to better define the relationships between a performer and the various types of data it handles, its association with the `CircularBuffer` class has been specialised accordingly. A performer therefore holds four sets of buffers:

- Inputs, are buffers that hold readings that are used as input by a performer From the performer point of view, these are read-only buffers.
- Outputs, are buffers that can hold any `Data` subclass in which a performer stores the outputs of its acquisitions or elaborations, so from its perspective they are write-only and the types of data written will depends on the nature of the performer.
- Commands, are buffer in which only commands are allowed and are used to send commands to a specific performer. From the performer point of view, these are read-only buffers.
- Alarms, on the other hand, only holds alarms and are used by the performer to notify whoever may be interested of an abnormal situation; they are usually write-only by the point of view of the performer that creates them.

Since a performer may receive inputs and commands from more than one other performer, each one has a set of input and command buffers.

Similarly, each performer may have a number of alarm buffers (the buffers on which it writes the alarms it generates) and output buffers. The need for more than one output buffer per type tracks back to one of the assumptions presented in Section 3.3.2, specifically the one about memory management responsibility: performers who *pop* information from a buffer are in charge of handling it and of its correct disposal.

This decision implies that once an information is popped from the buffer, the data structure does not retain a reference to the removed data, losing in fact the ability to access it and handle its memory deallocation. On the other hand, a performer that reads data from a buffer is in charge of deallocating it once it has made its computations and produced its output (if any).

The fact that each performer can (and should) deallocate the memory of the data it reads, means that if more than one performer needs to reason upon the same data, it should be duplicated and placed on different buffers, otherwise a performer could deallocate or modify an object before a different performer has finished using it.

Buffers therefore act as blackboard that allows to link different performers together and share data in a producer-consumer scheme. The same buffer will be identified as an *outputBuffer* from the producer and as an *inputBuffer* from the consumer point of view.

Since the design of the TDSH is focused on embedded platforms, memory optimisations have been considered: as an example, the inputs buffers of the performer are actually the same one of the performerTask, in order to prevent superfluous memory usage.

Once defined how the performer class handles data, it is necessary to clarify how this data is accessed and stored by a performerTask. As pictured in Figure 3.23 the performerTask has a reference to the inputs buffers: that is because it could benefit from knowing from which buffer each information comes from. In all the other situations, the performerTask would not benefit by facing the complexity of the buffers arrays and for this reason such cases are handled differently.

The output of data is performed by the `writeData` operation: the presence of multiple buffers is transparent from the task perspective, it will be its containing performer duty to handle eventual multiple data distribution inside numerous buffers. the same goes for alarms that are stored by the performerTask by means of the `writeAlarm` operation.

On the other hand, commands are an input buffer and for such reason, to simplify their handling by the task, it has been defined the `handleCommand` function: it takes a single command as a parameter and is invoked on the task on each command received by the container performer before executing the actual `perform` of the task, as shown in Figure 3.26.

As described in Section 3.1.4, the write and read actions of shared information should be synchronised with the activation times of the producer and of the consumer respectively. Since one of the assumption of TAM is that a time grain

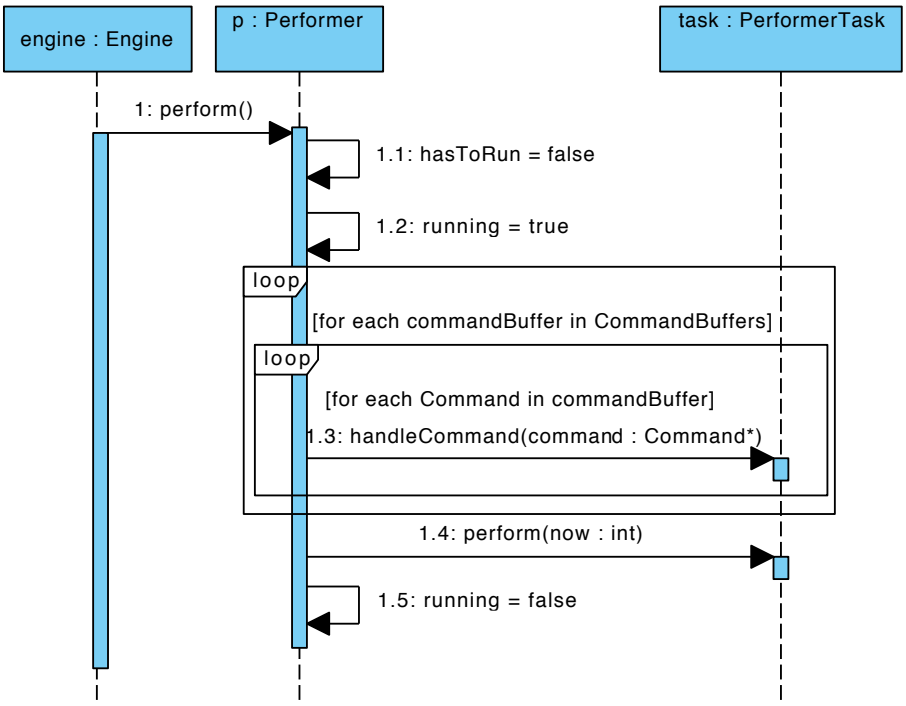


Figure 3.26: Perform operation.

is atomic and indivisible, anything that happens at a given time grain cannot be stored within the same grain as this would violate the atomic property. For this reason all the writings on the buffers are synchronised at the end of a "tick iteration", which means that information is written after all the timers have been ticked and all the performers that should activate at such instant have been executed.

This approach enforces data consistency between performers triggered at the same grain as another assumption of the TAM model is that all performers that are executed at the same grain must be perceived as simultaneous.

Consider the example scenario previously introduced and that at a specific time instant `t`, both `P1`, `P2`, and `P3` have to be executed. The execution order of the performers is virtually casual and should not affect the outcome of the system. Consider the execution order 1)`P2`, 2)`P1`, 3)`P3`: in such case `P2` would read the previous value produced by `P1` as valid for the instant `t`. After `P2` ends `P1` updates values in `B1` and `B2`. Finally `P3` is executed and reads the new value as the value for the instant `t`. This results in having two different values as output from `P1` at the same instant, which not only is theoretically wrong, but may also lead to practical issues.

This kind of issue is avoided by using temporary buffers within the `write*`

functions and defining an `expose` inside the performer class that moves the information from the temporary buffer into the buffers set for outputs and alarms. The logic of the expose oepration is pictured in Figure 3.27

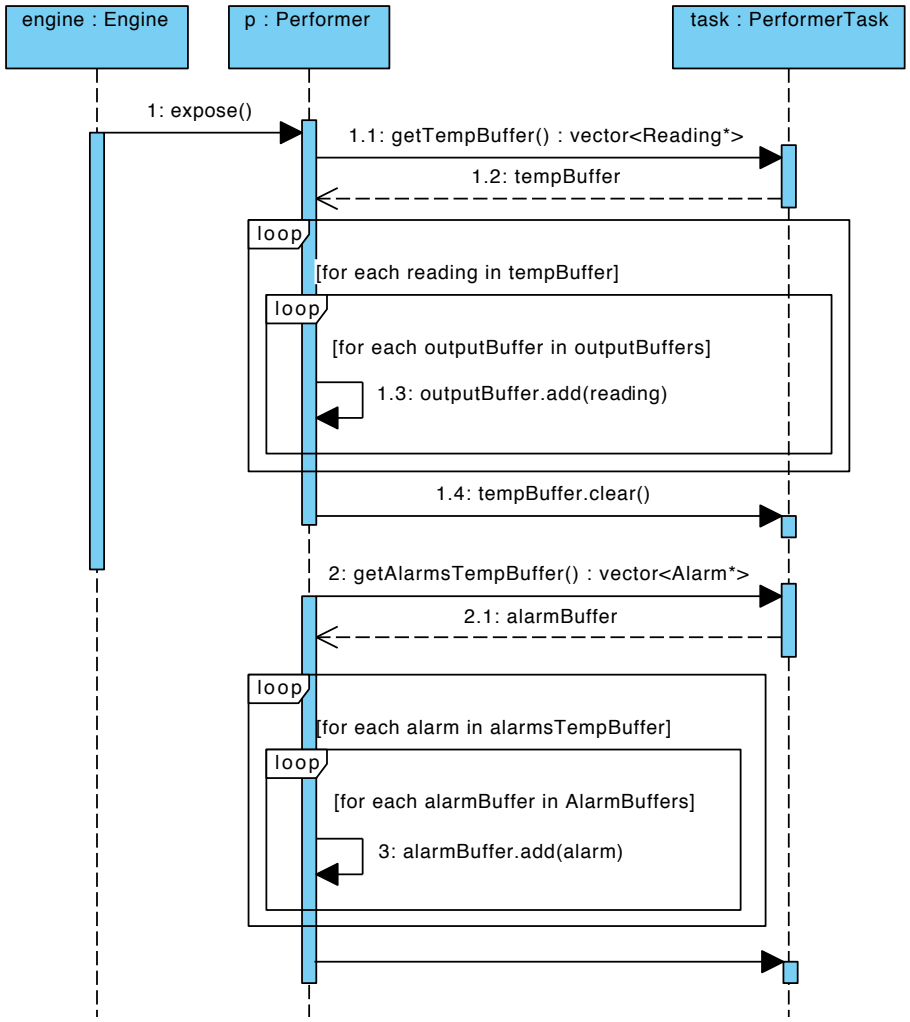


Figure 3.27: Expose operation.

The expose operation is called at the end of every execution loop of the system and is triggered by the engine on every performer that has been executed. The performer gathers the new information produced by the task and placed into the temporary buffer and writes it into each performer output buffer, then clears the temporary buffer and repeat the same actions on the alarms temporary buffer.

3.4.3 Engine

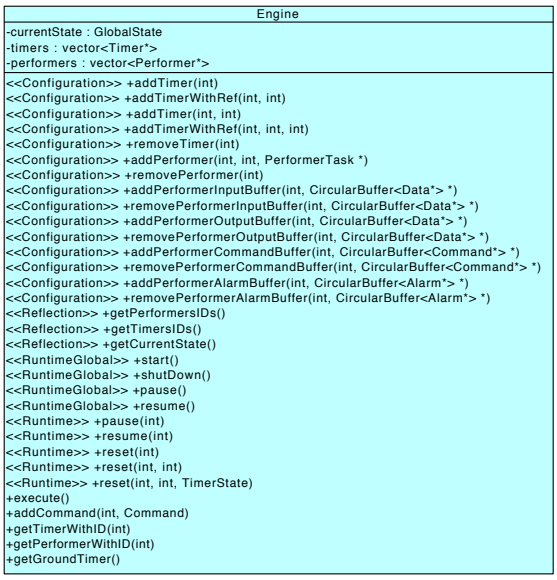


Figure 3.28: Concrete design of the Engine.

`Engine` is the entity in charge of coordinating the access and modification of the TDSH configuration. As pictured in Figure 3.28 the engine holds references to all the performers and timer, but is still a passive component. Every time the ticker triggers the ground timer, it in turn starts an update cycle by ticking its sub-timers and calling the `execute` function on the engine (as pictured in Figure 3.22). The `execute` operation itself is pictured in Figure 3.29: firstly it checks which performers need to be activated by means of the `hasToRun` flag that is set as shown in Figure 3.21, then it launch the `perform` operation on them (see Figure 3.26), and finally exposes any new data produced by the performers that have been executed by calling the `expose` operation on them (Figure 3.27).

3.4.4 Reflection and Configuration

TDSH configuration can be seen as a two fold procedure:

- Topological configuration: In TDSH topology is considered as the definition of the hierarchical structure of timers and how eventual performers are connected to such timers.

- State configuration: the state configuration is the set of values assigned to each timer in terms of period and currentState, the latter represent the state (running or suspended) in which each timer will be initialised.

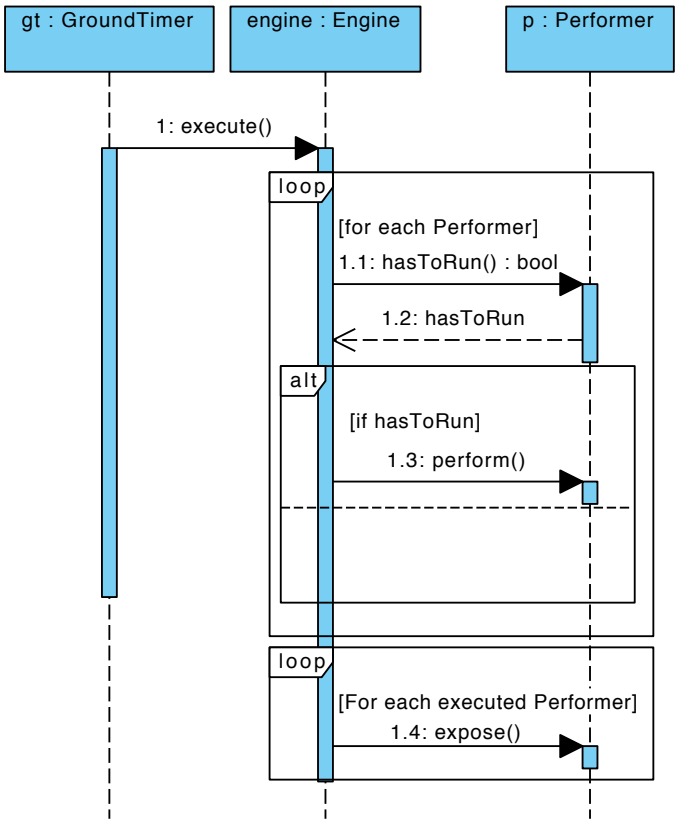


Figure 3.29: Execute operation.

Topological Configuration

The topological configuration, is again a two step process: first there is the definition of the structure in descriptive terms, such as a JSON file describing the timers tree and their performers; secondly there is the actual allocation of the data structures or objects derived by the textual description.

The definition of the description can be done by hand or a dedicated tool may be created. But in both cases it is reasonable to suppose that it is generated outside the micro controller and on a more user friendly environment (such as a tablet or a desktop machine).

The allocation of the hierarchical structure can be fulfilled with two approaches:

1. Can be statically written inside the micro controller firmware and loaded with it by a desktop machine.

2. Can be performed by the micro controller itself during boot.

In the former case, the descriptive file will be used by an interpreter that will instruct the assembler in order to obtain the right structures initialised. In the latter instead, such file may be loaded onto the micro controller that while booting will follow it and initialise the structure.

The on board initialisation represent a more modern approach, also given the fact that today micro controllers feature enough computational power to interpret strings. For that reason, it is the approach pursued in this work. The entity in charge of the configuration has been named `Manager`.

The manager must initialise and connect every timer, performer and buffer, it may do so by reading the structure from a file. In order to expose the configuration features that the manager exploits, an interface named `TDSHInterface` defining such operations have been designed and is pictured in Figure 3.30.

Domain applications and other components use the primitives defined in the interface and exploited by the manager in order to control the whole framework based on the configuration and external input (actual communication is provided by a dedicated communication performer).

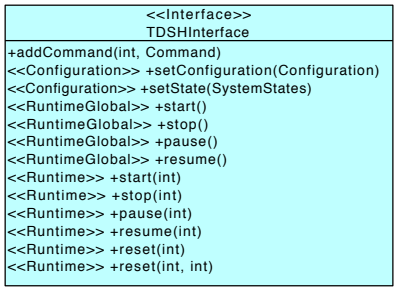


Figure 3.30: The TDSH Interface.

In turn, the manager relies on the engine as handling point of the single TDSH components as it exposes a set of functions aimed at adding and removing both timers and performers (aka it acts as a factory); it also allows the linkage of buffers to performers.

To enable timers to have sub-timers and performers, functionalities like `linkTimer`, `linkPerformer` and `addInputBuffer` have been introduced in the timer and performer classes.

All the configuration functions are labelled with the `<<Configuration>>` stereotype in all the components.

State Configuration

The basic state configuration can be achieved by the `reset` functions defined in timers and explained in Section 3.4.1. The engine exploits those functions and expose a similar set that allows the choice of the specific timer to initialise.

Reflection

At runtime, an application may be interested in knowing the current status of the framework, for that the `<<Reflection>>` functionalities have been added to all the entities.

For example, a timer exposes information about its current state, its reference timer, its sub-timers and the list of its performers. Note that timers and performers are returned as lists of IDs. Similarly the engine makes available information about the global state and the lists of performers and timers currently in the system.

3.5 Implementation

In this section, the implementation of the various components of TDSH and the structure of the TDSH Library is discussed.

3.5.1 Implementation Choices

The TDSH frameworks aims at being deployed onto modern micro-controllers, which means that it has to take into consideration the reduced resources available, but can still expect reasonable computational power and communication capabilities.

As a reference point with regard to a plausible platform a number of choices have been considered, and finally the STM32F4 boards family from STMicroelectronics[5] have been chosen, as mentioned in Section 3.2.2. The programming language of choice was C++, as it is cross-platform and supported by the vast majority of the modern micro-controllers (STM32F4 boards included), while offering the benefits of the Object-Oriented Programming paradigm.

The reference Integrated Development Environment (IDE) for developing on STM32F4 boards is Atollic Studio, a commercial IDE based on Eclipse. However it is possible to configure the environment also for Eclipse itself, which being free and opensource has been preferred to its commercial counterpart.

3.5.2 The TDSH Components

The implementation of the TDSH has been divided into two main components: the `TDSHLib` and the `STM32F4-TDSH`, with the former representing the core, device agnostic part of the framework, while the latter is strictly dependent on the hardware of choice.

TDSHLib

The TDSHLib library pictured in Figure 3.31 represents the core of the TDSH.

[5] www.st.com

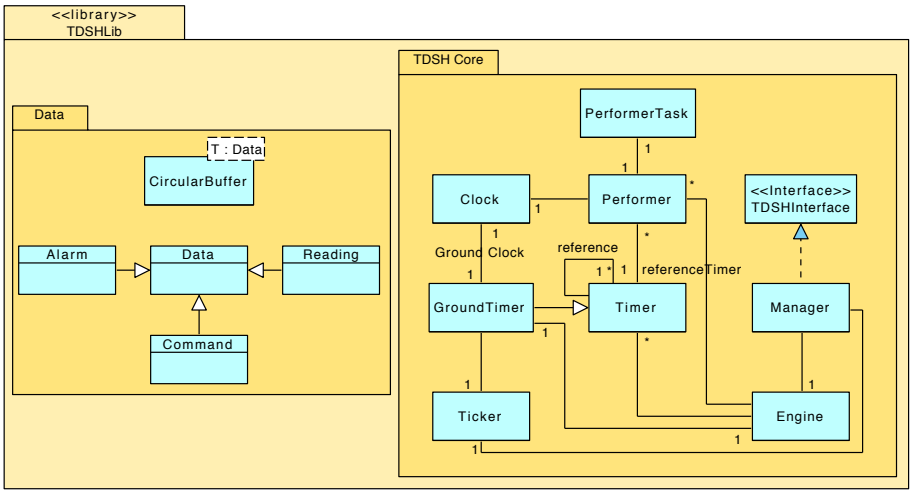


Figure 3.31: The TDSHLib Library.

The library is divided in two main packages: `TDSH Core` and `Data`. `TDSH Core` contains the core classes that reify the components introduced in Section 3.4.

The behaviour of the components is consistent with the description given, with the only restriction of the Manager class, which currently only holds a static configuration that must be arranged at code level prior to loading the firmware onto the board for execution and initialises the framework as pictured in Figure 3.32.

The primitives for proper configuration mechanisms and dynamic reconfiguration of the topologic structure of the framework have been included and such functionalities are under development.

The `Data` package instead contains all the basic models for sensor readings, commands, and alarms, as well as the circular buffer class used to store and share the information.

STM32F4-TDSH

Figure 3.33 represents the hardware specific components. The `STM32F4-TDSH` library is built upon the `TDSHLib` and exploits some of the hardware characteristics of the *STM32F4 Discovery* board. Figure 3.33 shows the main packages in this library

In the `Core` package the classes of the `TDSH Core` package from the `TDSHLib` library are specialised for the hardware. The `STM32F4Ticker` class is a specific kind of `Ticker` that exploits the `Systick` as a source for generating ticks.

The `Systick` is one of the internal timers of the ARM cortex-m4 cpu that powers the board that counts down from a reload value to zero, it wraps when it reaches zero and it does not decrement when the processor is halted for

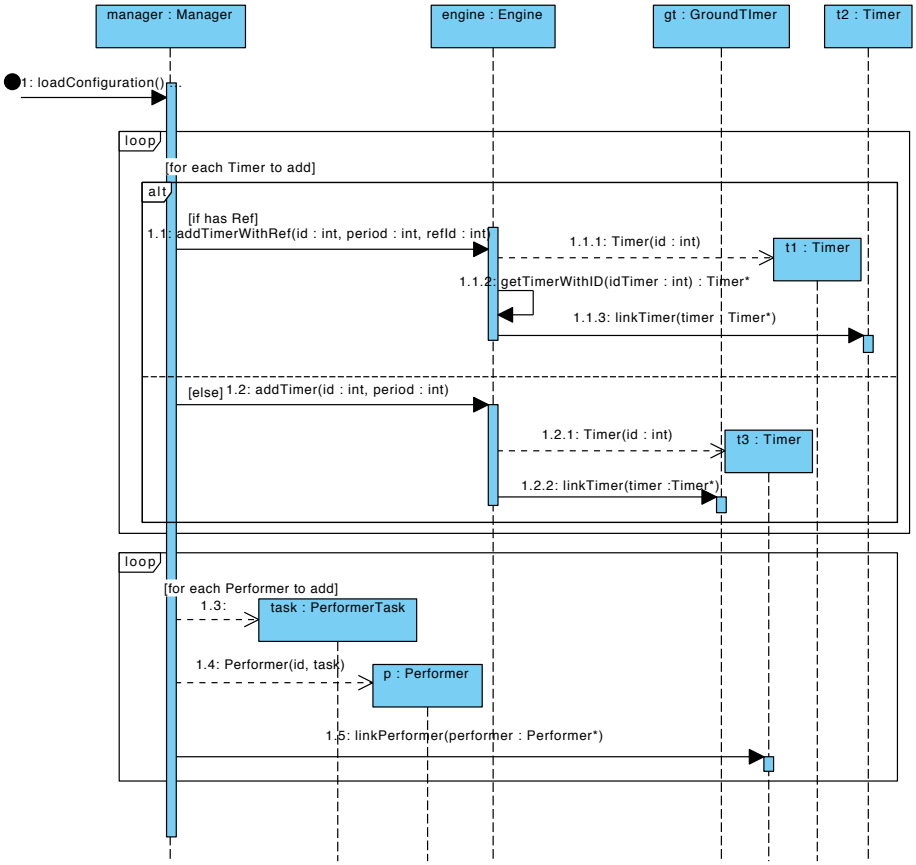


Figure 3.32: The configuration of the environment.

debugging. It is often used for producing the main system event clock. The timer speed is directly dependent on the cpu clock speed and it is configured as shown in Listing 3.1: the function `HAL_InitTick` initialises the systick timer in order to obtain a resolution of 1 ms and sets the priority of its interrupt.

In case of use of the standard `HAL_Delay` function from a peripheral ISR (interrupt Service Routine, or Interrupt Handler) process, it is important for the systick interrupt to have an higher priority (numerically lower) than the peripheral interrupt, otherwise the caller ISR process will be blocked. The function is defined as `__weak` in order to be overwritten in case of other implementations. The 1 ms period is obtained by invoking the system tick interrupt `SysTick_IRQn` every `HAL_RCC_GetHCLKFreq/1000U` clock cycles, where `HAL_RCC_GetHCLKFreq` returns the current cpu speed. This init function is called every time the cpu speed is modified in order to keep the timer consistent.

Listing 3.1: The Systick Configuration

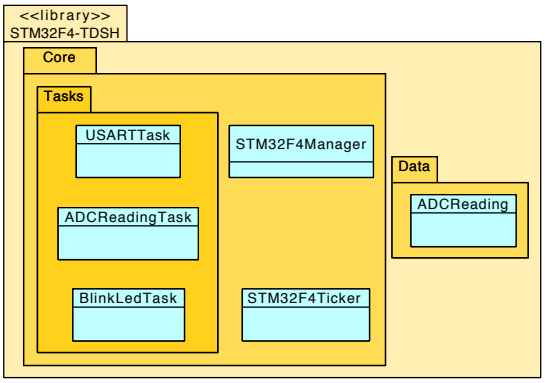


Figure 3.33: The STM32F4-TDSH Library.

```
1
2  __weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
3  {
4    /*Configure the SysTick to have interrupt in 1 ms time basis
       ↪ */
5    HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000U);
6
7    /*Configure the SysTick IRQ priority */
8    HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority ,0U);
9
10   /* Return function status */
11   return HAL_OK;
12 }
```

Once configured the systick timer, there are two ways to exploit its functionalities. The first one is to use its interrupt handler to invoke the `tick` operation on the ticker, but there are some issues to this approach. As mentioned the interrupt handler needs to have a very high priority, which means that it could be called while the engine is still executing the performers interrupting them and starting a new iteration, not only not being able to recognise the issue of a missed deadline, but also potentially leaving the framework in an inconsistent state.

The approach that has been followed instead is to let the counter related to the systick timer be incremented by the interrupt handler and then checking its values between different runs of the ticker. This solution, shown in Listing 3.2 is more robust in terms of general consistency: a framework run is always terminated (watchdogs can be configured to detects infinite loops and stuck components), moreover in case of a missed deadline, the amount of delay can be obtained. In case of Listing 3.2 a single delay put the entire board in an error state, but more complex behaviours can be modelled.

Listing 3.2: The Ticker run operation

```
1
2  void run()
3  {
4      uint32_t tickstart = 0U;
5      while(true){
6          /*Get the current value of the Systick timer*/
7          tickstart = HAL_GetTick();
8          /*Start the framework iteration*/
9          tick();
10         while((HAL_GetTick() - tickstart) < frequency){
11         /*Wait after the end of a tick iteration
12         before starting the next one*/
13         }
14         if ((HAL_GetTick()- tickstart) > frequency){
15             /*Enter a permanent error state with error led
                   ↪ blinking*/
16             while(true){
17                 BSP_LED_Toggle(LED5);
18                 HAL_Delay(100);
19             }
20         }
21     }
```

The Driver Library

As mentioned, the `STM32F4Ticker` class is hardware dependent and relies on hardware specific components such as internal cpu timers. In order to access such resources, microcontrollers are usually provided by their makers with low level libraries and functions.

When developing for a STM32 microcontroller there is currently a dispute about which library should be used as there are in fact, at least three choices that one could consider.

CMSIS CMSIS stands for Cortex Microcontroller Software Interface Standard. CMSIS is an hardware abstraction layer for Cortex-M devices like the one in use. This means that among different devices equipped with Cortex-M cpus there should be no issue about compatibility. However, communicating to peripherals such as SPI (Serial Peripheral Interface) can be cumbersome as CMSIS is very limited and most peripherals are excluded. Moreover communication with peripherals requires a direct registry manipulation, which means that is a very error-prone approach. It however enables to write the most efficient software.

SPL The Standard Peripheral Library is a set of libraries created by ST with the aim of simplifying programming for their units. However learning the use

of the library is not straightforward as the company published a lot of examples on which programmers can learn but a very scarce and not well organised documentation. Moreover in order to understand it and use it in practice some knowledge about the architecture of ST microcontrollers is needed.

Finally it is known to have quite a number of bugs that are not likely to be fixed (is being discontinued) and there are some complex situations (such as more peripherals depending on each other) that are not considered and need to be handled manually operating directly on the registers, which can be done, but removes the only true advantage of the SPL library that is code portability.

HAL Compared to SPL, the Hardware Abstraction Layer has more features. It was rebuild from the ground up and is completed with a tool for automatic code generation: the *STM32CubeMX* (Windows only). HAL delivers the possibility of reuse the same code across different chip families based on Cortex-M core from ST: the is a set of libraries, one for each chip family, which have (almost) identical API. This allows to run the same code on a different micro controller just by switching to the new library.

There are of course limitations, but it makes HAL quite versatile and reliable. As an addition with respect to SPL, HAL is not only a set of libraries for the internal peripherals: it allows to configure *FreeRTOS* (a real-time Operating System) and FatFS (File System) and SD card support along with it. Finally there are additional libraries built on HAL that contain sets of drivers for external devices like LCDs, sensors and more.

LL While HAL is currently receiving great support and bug-fixing from ST, there is a recent introduced third option: Low Layer. LL can be used in conjunction with HAL and STM32CubeMX should allow the choice between the two for each peripheral in use. LL uses less resources but the resulting code is less portable and it is not currently supported through out all STM microcontrollers families. LL is itself divided into three levels of APIs: low level, middle level, and high level.

Low level is for direct register operations, while middle level is used for tasks such as setting flags in peripherals registers such as ADCs or timers. The high level of LL is similar to HAL, it is responsible for configuration and initialisation of peripherals.

The bottomline is that LL is more tailor-made in comparison to HAL, which means that is less portable, but it has a very high optimisation level.

TDSH uses the HAL library as SPL was being discontinued and LL was not yet available. Figure 3.34 shows how the STM32F4-TDSH library is built upon the core TDSHLib and how it uses `STM32F4-HAL-LIB`, which is the implementation of the HAL library for the STM32F4 family of boards.

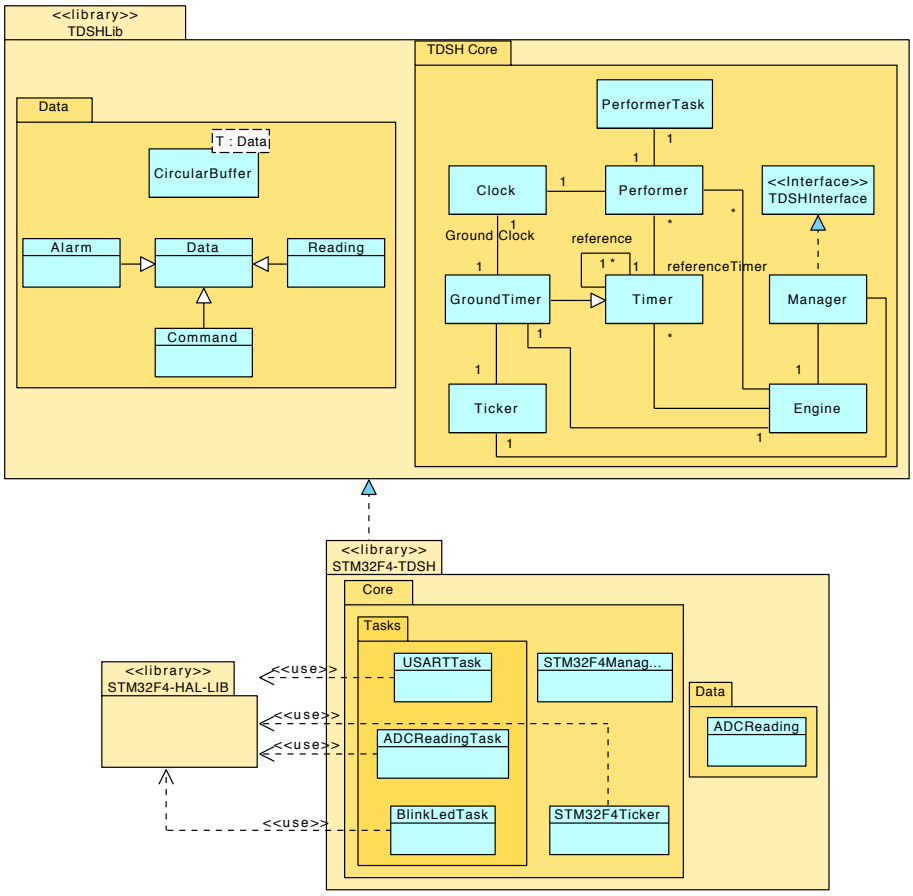


Figure 3.34: The TDSH set of libraries.

3.5.3 The System Overhead

The Overhead introduced by the TDSH framework is due to the additional computation needed to update timers and to go through all performers to execute. The operation that mainly constitutes such computation is the `tick` of the ground timer (and the ones of the descendant timers) and the `execute` of the engine called by it.

Listing 3.3: The GroundTimer tick

```
1 void GroundTimer::tick(){
2     if (_status != RUNNING)
3         return;
4     _groundClock->tick();
5     Timer::tick();
6     _engine->execute();
```

```
7 }
```

The `tick` of the ground timer pictured in Listing 3.3 is computationally equivalent to the general timer `tick` shown in Listing 3.4 as apart from the execute its differences from the general operation are constant in complexity.

Listing 3.4: The Timer tick

```
1  void Timer::tick(){
2      if (_status != RUNNING)
3          return;
4      _internalCounter++;
5      if((_internalCounter % _period)==0){
6          _internalCounter = 0;
7          emit_event();
8      }
9  }
10
11 void Timer::emit_event(){
12
13     for(int i=0;i<timers.size();i++){
14             timers[i]->tick();
15         }
16
17     for(int i=0;i<performers.size();i++){
18         performers[i]->setToRun();
19     }
20
21 }
```

as shown in Listing 3.4, the complexity of the `tick` is due to the `emit_event`, which loops between the direct descendants and call their `tick`. Similarly there is one `setToRun` invocation for each performer of the timer. The worst case is represented by the situation in which at every run all the timers and performers are ticked and executed, which means that both operations are called at worst once for each timer at each run of the ticker. Since the cost of the `setToRun` is constant, the final complexity of the `tick` for the ground timer is linear and equals to $O(T+P)$, where T and P respectively represent the number of timers and performers present in the system.

Listing 3.5: The Engine execute tick

```
1  void Engine::execute(){
2      for (auto iterator : _performersMap){
3          if (iterator.second->hasToRun()){
4              if (iterator.second->isRunning()){
5                  //DEADLINE MISSED FOR THE PERFORMER
6              } else{
7                  iterator.second->perform();
8              }
9          }
```

```
10      }
11      epilogue();
12  }
13
14  void Engine::epilogue(){
15      for (auto iterator : _performersMap){
16          if (!iterator.second->isRunning()){
17              iterator.second->expose();
18          }
19      }
20  }
```

By not considering the actual `perform` complexity, which depends on the task implemented and is not part of the overhead, the `execute` operation in 3.5 is, again, linear with respect of the number of performers of the system that have to be executed. The worst case is then $O(2P)$. In this analysis the cost of the `expose` operation has been assumed constant as no matter how many data there are to be exposed, memory to memory data transmission can be achieved using Direct Memory Access and thus not consuming cpu power.

With the given assumptions, the global overhead of the framework is then linear in the specific form of $O(T + 3P)$ and is split part (H in Figure 3.35) before the execution of performers (E_p) and part after (H') and after.

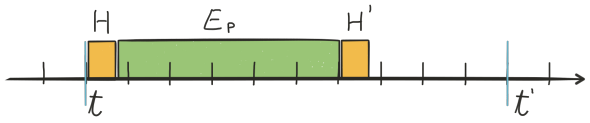


Figure 3.35: Execution time example.

Performers Durations Constraints

As mentioned in Section 3.3.1, the current implementation shows a restriction in terms of computational time for the performers execution. Specifically, the sum of the execution time of all the performers plus the overhead added by the framework, must be smaller than the duration chosen for the most fine time grain present in the system, which is equal to the period of ticker events. The reason is quite straightforward: the deployment platform is equipped with a Cortex-M4, a single core ARM processor and the environment is single threaded, which means that every run of the framework must take less than what is defined as a grain of the ground timer from the ticker.

As an example, Figure 3.35 showed the general structure of the execution time division between among the framework. Defined as t and t' two consequent ticks from the ticker, it is clear how every performer execution time must be negligible with respect to the time interval $t \rightarrow t'$. Particularly, in Figure 3.35, the portion of overhead related to the tick of every timer in the framework and the check of the performers that are to be executed is denoted as H, the sum

of the execution time of every activated performer is marked as E_p and H' represents the overhead related to the expose of newly produced information within the framework.

As mentioned E_p represents the exact sum of the execution time of performers, as while from a framework point of view their execution is simultaneous (in terms of frameworks ticks), the underlaying hardware and software does not allow any kind of true parallelism, making every performer being fully executed in a serial way.

Given how performers are executed and how the framework burden is structured and spread at runtime, it is now clear how the sum of every performer and the introduced overhead must be less than $t \to t'$.

3.6 Acquisition Case Study

As mentioned in Section 2.2.1 the validation for the TDSH component uses the case of fall detection as application domain. Figure 3.36 shows the structure of the case study. The final deploy is composed by three distinct modules:

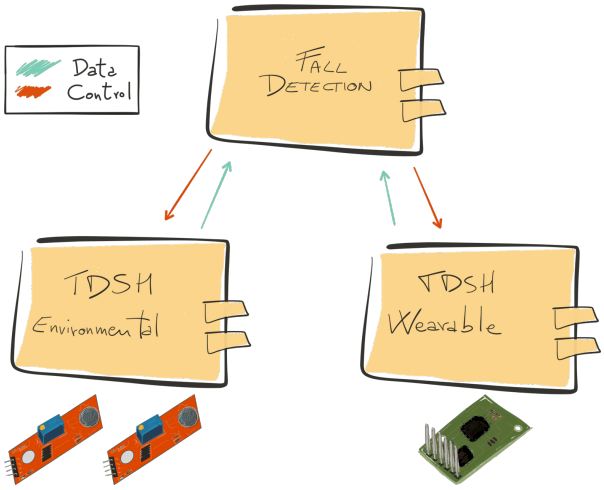


Figure 3.36: Case Study Structure.

1. The wearable node, which hosts an instance of TDSH configured to acquire accelerometric data from a physical accelerometer mounted onboard the STM32F4-Discovery board.
2. An environmental node, which also hosts an instance of TDSH configured to read from a linear microphonic array composed by two electret microphones.
3. A central node, which hosts the data fusion component and the fall detection component.

3.6.1 Wearable Accelerometer readings

The sensor

The STM32F4-Discovery board features an embedded accelerometer, which has been used for the purpose. It is a LIS302DL ST MEMS 3-axis accelerometer: it features dynamically user selectable full scales of $\pm 2/\pm 8$ and it is capable of measuring acceleration with an output rate of 100 Hz to 400 Hz.

Acquisition Sampling rates

The chosen sampling rate is 50 Hz, such value has been chosen in order to be consistent with the literature [23]. Lower speeds are possible, but rapid changes in acceleration may be missed leading to classification mistakes. As an example, Figure 3.37 pictures a simulated fall acquired at ~ 50 Hz.

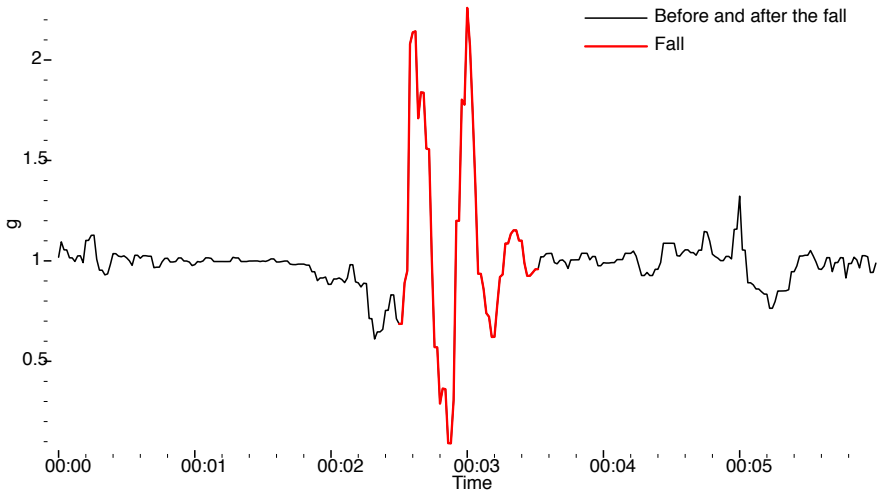


Figure 3.37: Magnitude of a simulated fall at 50 Hz.

Sampling Sizes and Data Transmission

Table 3.3 reports the dimensions of an accelerometric packet that contains the three dimensions of acceleration plus a timestamp. Using directly the magnitude in the form of $\sqrt{x^2+y^2+z^2}$ would reduce the size of the data produced to $\sim \frac{1}{3}$, however by keeping the original signal for data collection may allow the computation of different feature vectors.

Data transmission happens by means of an USART (Universal Synchronous-Asynchronous Receiver-Transmitter) interface. Standard transmission rates for this kind of communication are around 115.2 Kbps. However, in order to free the user from a wired connection from the wearable device to the central node,

Table 3.3: Accelerometer data sample

Value	Dimension	Unit
TimeStamp	16	bit
Acc_X	16	bit
Acc_Y	16	bit
Acc_Z	16	bit

Table 3.4: TDSH timers settings for the wearable node

TimerID	ReferenceID	Period	Period (ms)
GT	-	1	10
VT1	GT	2	20
VT2	GT	50	500

a wireless transmission medium should be used. The standard speeds for ad hoc wireless transmission interfaces are lower than standard USART, in the range of 57600 Kbps, which is nonetheless enough for the purpose, being the data production rate of $(16+16+16+16)*50 = 3.1Kbps$. As transmission protocol Bluetooth has been chosen over Zigbee as now available in a number of devices, included smartphones, thus making it possible to receive data from a standard smartphone for future developments.

The RN42 is a small form factor, low power, Class 2 Bluetooth radio. It can deliver up to 3 Mbps data rate for distances up to 20 meters. It supports a number of protocols, USART included, which means that once configured it can be physically linked to the pins of the board and a serial connection can be established from the central node as if it was wired.

The TDSH structure

Figure 3.38 shows the TDSH configuration for the accelerometer based wearable node. The timing relations between the timers are described in Table 3.4. There are two performers as follows:

- P_1 is the acquisition performer, it is in charge to read the samples from the physical sensor and it places them into its output buffer B_1, which is shared with P_2. It is activated by VT_1 every 20ms, resulting in a 50 Hz sampling rate.

- P_2 is a communication performer, it exploits a serial communication interface and streams packets from B_1 outside the board through a DMA transmission of type *Memory-to-Peripheral*. It is activated by timer VT_2 every 500 ms.

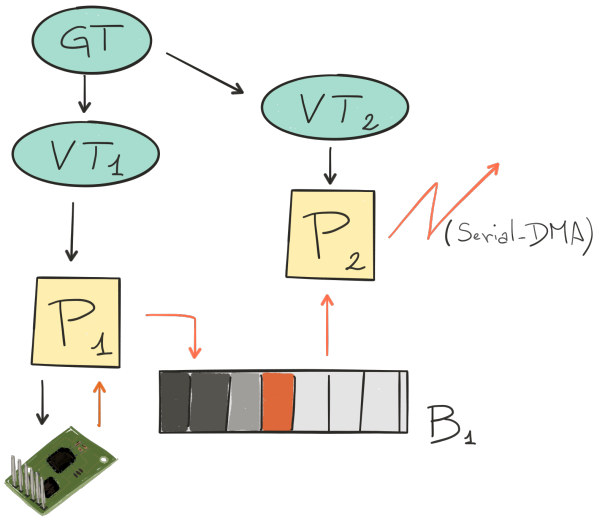


Figure 3.38: The TDSH structure for the Accelerometer node.

3.6.2 Environmental Microphonic Array

Here the deploy structure of the acoustic localisation module is presented. The TDSH framework is deployed on a STM32F4-Discovery Board and has a linear microphonic array attached to it composed by two electret microphones.

The microphonic array structure

The two microphones are placed at a predetermined distance between them along the vertical axe. The distance proposed is $\sim$ 1m, which is consistent with the literature of reference [76]. By knowing the relative position of the microphones, and the absolute position of the array in the environment, it is feasible to infer information about the direction of the source of sound events.

Specifically, the linear microphonic array is composed of two Keyes KY-038 modules, as the one pictured in Figure 3.39.

The outputs of the module are as follow:

D0 Is the *Digital* output, its value is 0 if the audio intensity is below a specific threshold, 1 otherwise. The threshold can be regulated by the onboard potentiometer.

A0 Is the *Analog* output, the intensity of the output is influenced by the potentiometer (gain).

Acquisition sampling rates

The sampling rates achievable depends on the quality of the Analog to Digital Converter available. In this specific case, three ADC converters are available, bit

Figure 3.39: The microphone unit used.

Table 3.5: Sampling rates and relatives frequencies

Sampling Rate (KHz)	**Maximum Frequency (KHz)**
08.000	03.6
11.025	05.0
22.050	10.0
32.000	14.5
44.100	20.0
48.000	21.8
64.000	29.1
88.200	40.0
96.000	43.6

resolution is equal to 12 bit in normal use situations. The conversion frequency can be tuned by acting on a pre-scaler embedded within the board firmware. ADCs on the STM32F4-Discovery can achieve ~ 2.4 MSPS (Mega Sample Per Second) and by interleaving all three ADCs and timing the acquisition it is possible to obtain 7.2 MSPS. At such high sampling rates, memory and speed issues arise, but such high speeds are not always required.

The frequencies obtainable from the microphones are directly dependent from the sampling rate: higher frequencies requires higher sampling rates (see Table 3.5).

44.1 KHz and 48 KHZ are the sampling rates that are usually used when acquiring audio destined to be listened by humans as they allow to cover the frequencies within the human audible spectrum. However, it should be not take as granted that the same frequencies are the ones that give the most significant data when coming to software systems. In fact, the sampling rates commonly used in the literature for audio analysis and events classification based on audio traces are between 8 and 16 KHz [57, 76, 77], which is why the sampling rate fixed for this experiment is 10 KHz.

10 KHz is a low sampling rate if compared to the max speed achievable by the ADC itself, however such rate raises a timing issue within the current implementation of the TDSH framework. As mentioned in Section 3.5.2 the accuracy produced by the framework is 1 millisecond. This means that for each activation of the framework there should be a performer able to acquire 10 values, one each 100 microseconds before its next run. But doing this would violate the assumption that performers work under the timing constraints and scales offered by the framework, so a different solution has been adopted: using the Direct Memory Access (DMA) option along with the ADC triggered by an hardware timer only dependent on the board clock. By adopting a pre-scaler, a timer has been configured in order to trigger at 10 KHz and DMA allows the ADC to write the values directly on an array, that can then be read in chunks by the performer that handles the ADC.

The ADC is therefore autonomous within each sequence of conversions. Such sequences could be manually triggered by the performer but a number of downsides have ben identified with this approach, namely: there could be a timing delay between the end of an acquisition sequence and the next one, as the frameworks ensure executions at a precision lower than the one of acquisition the conversions could end a number of microseconds before the performer start, causing a discontinuity within the data stream; moreover such discontinuity could easily be variable and not predictable.

For this reason a different approach was adopted: the ADC performs conversions in a *continuous* mode and stores the values in a buffer, which is double the size of the chunks read by each performer execution, as shown in Figure 3.40

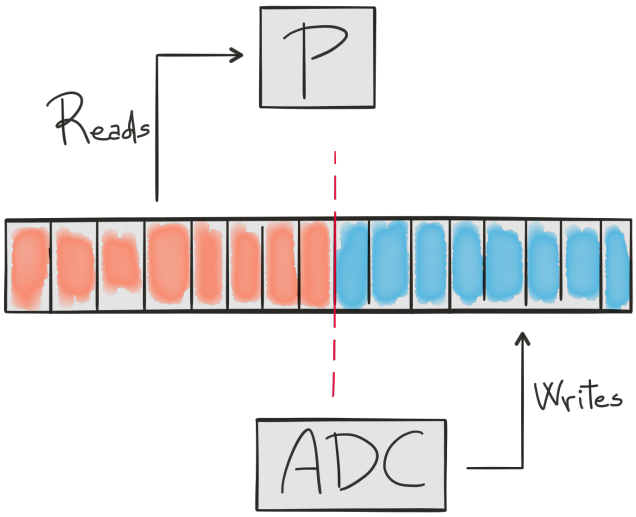


Figure 3.40: The continuous ADC setting adopted.

Using this configuration the ADC is fully autonomous and it is only started and halted manually. Discontinuities have also been eliminated as the ADC never stops between sequences of conversion, it starts populating one half of the buffer while the performer reads the other half.

Sampling Sizes and Data Transmission

Table 3.6 shows the sizes of a microphonic array sample. It is noteworthy that while the stated precision of an ADC sample is 12 bit, this value must be stored in a 16 bits variable (as the smaller alternative would be only 8 bits).

Table 3.6: Microphonic array data sample

Value	**Dimension**	**Unit**
TimeStamp	16	bit
Mic1	16	bit
Mic2	16	bit

As from the TDSH specifications, each sample has its own timestamp. Considering the sampling rate of 10 KHz, the size of data generated for each second S is as follow:

$$\begin{aligned} S &= (TimeStamp + Mic1 + Mic2) * 10.000 \\ &= (16 + 16 + 16) * 10.000 \\ &= (48) * 10.000 \\ &= 480.000\ b\ s \simeq 468\ Kb\ s \end{aligned} \tag{3.1}$$

In this example the data overhead introduced by timestamps represents $\frac{1}{3}$ of the data produced ($\sim 33\%$). It is however easy to reduce it to less than 1% of the data. By packing data into windows of $0.1\ s$ and relaxing the constraint of having a timestamp on each value the resulted amount of data produced each second (S') becomes:

$$\begin{aligned} S' &= ((Mic1 + Mic2) * 1.000 + TimeStamp) * 10 \\ &= ((16 + 16) * 1.000 + 16) * 10 \\ &= ((32) * 1.000) + 16) * 10 \\ &= 320160\ b\ s \simeq 312\ Kb\ s \end{aligned} \tag{3.2}$$

S' is calculated by providing an explicit timestamp only once per packet, specific timestamps for samples within packets can be easily calculated since the sampling rate is fixed and known.

Being able to actually analyse and transmit this amount of information represents different issues. As for data transmission, being the node environmental,

Table 3.7: TDSH Timers settings for the environmental node

TimerID	ReferenceID	Period	Period (ms)
GT	-	1	1
VT1	GT	10	10
VT2	VT1	10	10
VT3	VT1	10	10
VT4	VT1	10	100
VT5	VT1	10	100

a wired connection is allowed and exploited. As mentioned, standard transmission rates for serial communication go up to 115.2 Kbps, which is notably less than the 312 kbps previously calculated. Modern interfaces however support higher transmission rates, so the serial communication speed has been set at 460.8 Kbps that allows to avoid a bottleneck inside the board at runtime.

The TDSH structure

Figure 3.41 shows the chosen configuration of the TDSH framework in order to deploy the microphonic array node.

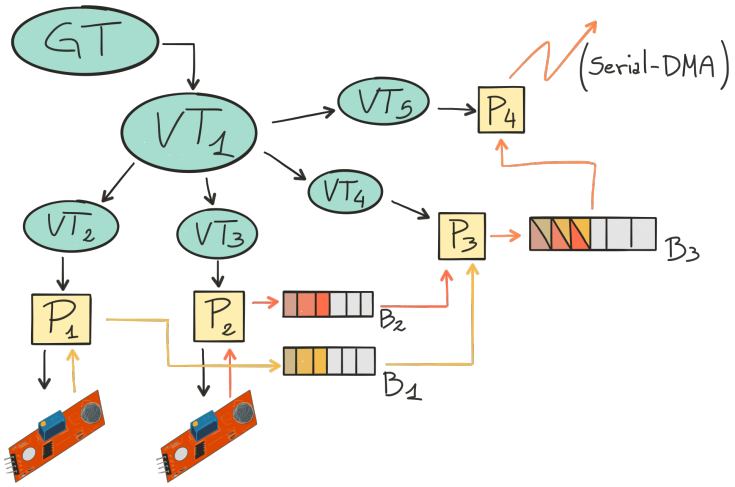


Figure 3.41: The TDSH structure for the microphone array.

Table 3.7 also shows the configuration of the module, along with the timings for each performer:

- P_1 and P_2 are the performers in charge to read the ADCs. Their activation timers are VT_2 and VT_3 respectively, they are sub-timers of VT_1 to be able to change both activation times just by acting on their reference

timer. The values they read are placed in buffer B_1 and B_2 for P_1 and P_2 respectively.

- P_3 produces the combined packets, with timestamps and readings from both microphones, effectively creating the objects as described in Equation 3.2. It has B_1 and B_2 as input buffers and put its outputs in B_3.
- Finally, P_4 represents the transmission performer: it features a serial connection and sends the packets of data outside the board through DMA transmissions of type *Memory-to-Peripheral*.

3.6.3 Fall Detector

The application node is developed in a desktop environment and it is not built on TDSH. Specifically, it is a quite straightforward software component built in Processing[6]. The choice of the language is due to the embedded and direct support to hardware communication interfaces such as USART. Since they are not a crucial part of the setup the detection algorithms are basic and should be swapped with more complex counterparts in a real system.

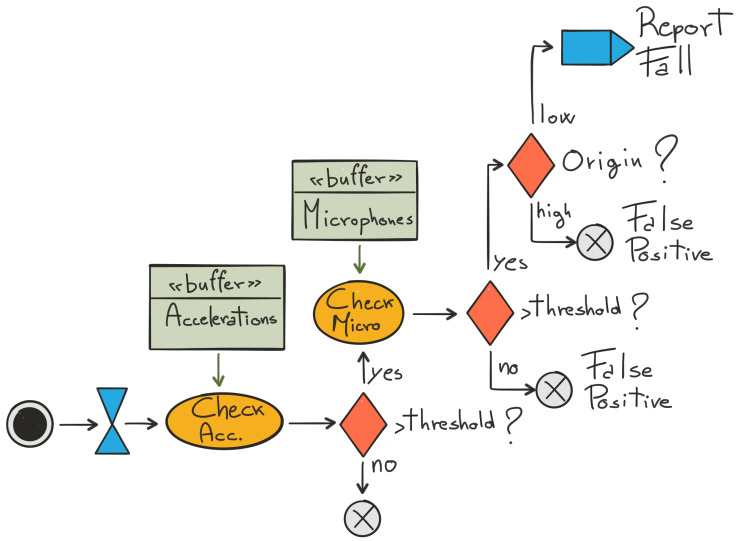


Figure 3.42: The fall detection algorithm.

Figure 3.42 reports the working behaviour of the fall detection algorithm:

- The component checks the accelerometric data for possible falls.
- If any activity over the predefined threshold is detected the microphonic information is checked. The threshold is set to 1.5g as in [23].

[6] https://processing.org

- The microphonic values are checked in a window of 0.5 seconds centred in the time of maximum acceleration, if a loud sound is detected then the origin of the sound is evaluated. The threshold for the loudness has been empirically determined. In case no loud sound has been recorded in the same time window of the acceleration data the fall is flagged as a false positive.
- Finally, if the origin of the sound is labelled as *low* the fall is reported to the user, otherwise it is labelled as false positive. The origin of the audio event has been simplified and just keeps track of the higher mean intensity: if the lower microphone detected an higher mean intensity the event is labelled as *low*, *high* otherwise.

3.6.4 Discussion

The presented scenario was built in order to test the TDSH approach and its implementation. The wearable node presents a basic configuration, with a small number of performers and works at low speeds (10 ms). Deploying the communication performer presented some issues related to the use of DMA for transmitting the data. The usage of the structure pointed out some bugs and memory leaks within the implementation of the circular buffer, which lead to a rethink of the memory management mentioned in Section 3.3.2. Moreover the working principle of the ADC in continuos mode with DMA support initially produced gaps in the acquired data, due to the reading activity of the ADC inner buffer from the performer before it could fill it with new conversions. Usually these kinds of delays are negligible, but the high speed of the ADC readings forced the used of the *double* buffer solution, pictured in Figure 3.40.

The implementation of the microphonic component however exposed the limits of the implementation. The high speed required by the sensor (10 KHz) meant some optimisations in the acquisition process were needed: they are treated in Section 3.6.2.

The central node highlighted another issue: time synchronisation between different nodes. In the test application the wearable and environmental nodes were activated in sequence. Sequential activation however introduces a systematic difference between the two nodes time references. In this context, the delay was estimated and approximated as the time of execution of each transmission instruction, nonetheless a more careful analysis of this issue could be performed as a future development.

The experiment was run just for testing purposes, no real falls were performed and no data collection has been stored for further use. A more consistent trial could be performed in future works in the field of fall detection.

CHAPTER 4

Subjective sPaces Architecture for Contextualising hEterogeneous Sources

This Chapter describes the Subjective sPaces Architecture for Contextualising hEterogeneous Sources (SPACES). As mentioned in Chapter 2 the identification of a suitable set of architectural abstractions, able to represent sensor measurements independently from the hardware characteristics of the source, could improve reusability, openness, and modularity of software systems.

4.1 SPACES - The Underlaying Concepts

This section presents the general concepts of the SPACES model. SPACES is a set of architectural abstractions that removes the dependency from the sensor by contextualising (sensor) measurements in a spatio-temporal frame. In detail, measurements have a time-stamp and are localised in both a measurement and a positioning space. Such spaces are subjective to the software components that manage them.

For example, a sensor component that is in charge of acquiring temperature expressed in a Celsius measurement unit, will localise the sensed data in a Celsius measurement space and in a Cartesian 2D space that is local to the sensor.

Mapping functions allow to project localisations from what are defined as source spaces into target spaces (possibly different) that are in turns subjective to the software component interested in managing the sensed data in those target spaces.

Suppose in the previous example, that the system includes also another component that operates in terms of Fahrenheit degrees and that is in charge of controlling the fan coolers that are present in a room according to the tempera-

ture measured near the fan coolers. To exploit the acquired data, a conversion is required. Thus, a mapping function could be defined in order to map the sensor component data from Celsius to Fahrenheit for the measurement and from the local 2D Cartesian space of the sensor to a global 3D Cartesian representing the room.

The general concept is that any space can be considered as a source with respect to a target space. A mapping function is in charge of mapping a couple of spaces (measurement and positioning spaces) that acts as source in a couple of corresponding spaces that acts as target spaces. The pattern defined by the abstractions can be replicated many times, at different abstraction layer.

When acquired, the data is localised in the subjective space of the acquisition component, then, the data can be subjected to several incremental transformations according to the typology of spaces (both measurement and physical) known by the component that gradually have to manipulate the data.

The main benefit is that applications no longer need to know neither the type nor the numbers of deployed sensors. Upon the occurrence of an event of interest, applications can decide to access all the other events that are related both spatially and temporally. At this level of abstraction, the distribution, usage, and eventual storage of data are out of scope, as the focus is on the definition of the architectural abstractions that could solve the sensors heterogeneity issue, thus facilitating the handling of those activities.

The remainder of this Section will cover the concepts underlying SPACES with the following simplified case of study. Consider a smart building composed by different rooms; in each room different sensors are located. In this example, a single room (`room1`) is instrumented as follows: in the top corner there is a camera (`cam1`) facing the centre of the room. Hanged on the wall there is also a thermometer (`therm1`). Moreover, a person in the room owns a smartphone with the accelerometer `acc1`. In this kind of contexts, smartphones are usually considered as extensions of the user, which means that their position is the same. Several applications can rely on the above listed sensors: a tracking application could try to follow the user (either a specific one or any user) and could make the position available to the system; an application could exploit the locations of the users to control the temperature in the rooms accordingly, based on their needs or preferences. These are just a few examples that can benefit from the proposed approach.

4.2 Spatial Model

Spatial contextualisation derives from the concepts presented in [91]. Those concepts have been revisited and enriched in order to capture and represent the spaces as required so that a measurement can be fully described via spaces.

4.2.1 Core Concepts

A *Space* is defined as a set of potential locations, that are all the locations that could be theoretically considered in that space. In a graph, the potential locations are all the nodes and the edges. On the other hand, if a Cartesian space is used to localise entities within a room, then the potential locations are every point in $\mathbb{R}^2$ of the area delimited by the room perimeter. Applications, when dealing with a space, explicitly manage effective locations, which are a subset of a space potential locations. For example, an application that calculates the trajectory of a mobile entity will only explicitly consider a finite number of locations in the Cartesian space, that is, the locations belonging to the trajectory (the effective locations). Each space defines at least a *premetric*. The premetric defines the distance between two locations as a positive, non-zero number if the two locations are distinct, and zero if the locations are the same. These concepts are presented in Figure 4.1.

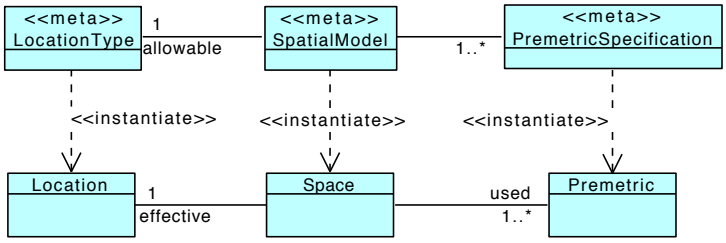


Figure 4.1: Core concepts, meta representations and corresponding instances.

Spaces and (effective) locations are respectively created from *Space Model* and *Location Type* as depicted in Figure 4.1: a space model thus specifies the type of allowable locations and at least one premetric that can be applied to a pair of locations.

4.2.2 The Concept of Dimension

The main extension to the original model consists in defining spaces as an aggregation of *Dimensions* as pictured in Figure 4.2. A Dimension literally represents a dimension in a space, where a space location is considered as an aggregation of dimension values. For example a location in a 3 dimensional Cartesian space, would be represented as an aggregation of 3 values, one for each dimension of such space. Since a space is defined as an aggregation of dimensions, a space model will be composed by at least one *Dimension Model* that exhaustively describes a dimension and all its components. Space models, location types, and premetric specifications are meta-level concepts that define the base-level concepts (spaces, locations and premetrics) applications deal with.

Moreover a Dimension is also characterised by a *UnitOfMeasure*, a *Precision* related to its values, and a set of *Boundaries* related to the values that each

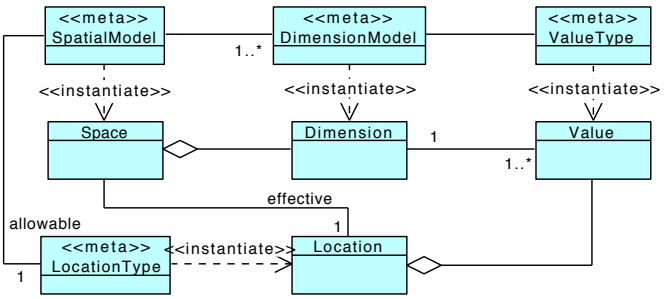


Figure 4.2: The concept of Dimension.

dimension can assume. The complete model of a dimension is pictured in Figure 4.3.

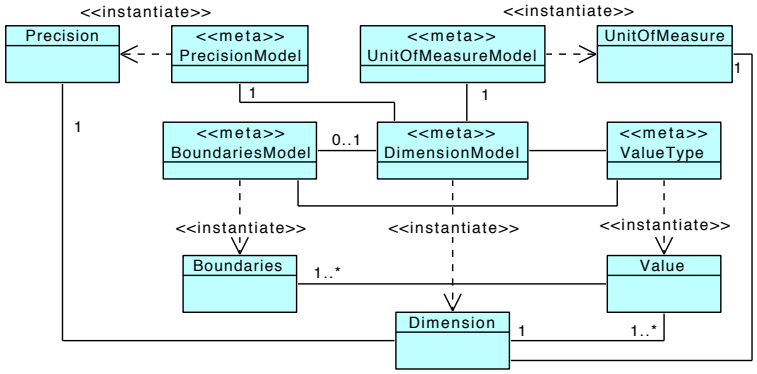
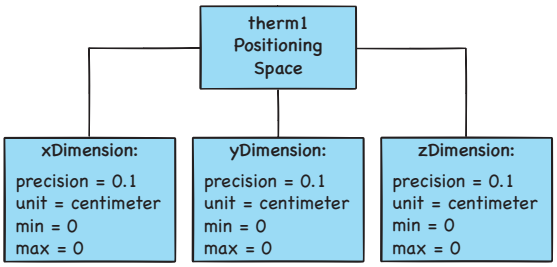
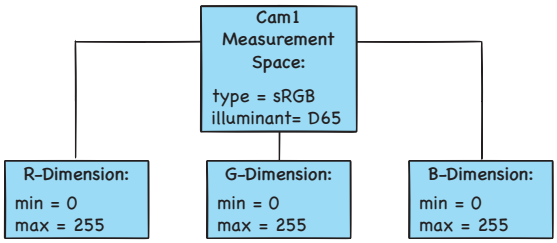


Figure 4.3: The Dimension elements.

Considering the example scenario introduced in Section 4.1, `room1` is represented by a 3D Cartesian Space (a positioning space). Suitable locations for this kind of space are 3D points. For example, Figure 4.4a pictures the three dimensional Cartesian space in which `therm1` may position its samples, composed by three dimensions (x, y and z). Each dimension has its own unit of measure and precision. Boundaries are expressed as values, each dimension has a *min* and a *max* admissible value. Similarly, Figure 4.4b sketches a measurement space example: it represents the colour space and the allowable type of values of the possible locations for `cam1`, which are the pixels of the images.



(a) 3D Cartesian positioning space example.



(b) RGB measurement space example.

Figure 4.4: Examples of spaces.

4.2.3 Zone and Membership Function

The concept of *Zone* has also been introduced. A zone Z_S is a subset of potential locations of a space S. It is defined by a set of effective locations termed *Characteristic Locations* in S and by a *Membership Function* that states if a given location of S belongs to the zone. Essentially, the membership function is a boolean function that is true when a location falls within the zone. According to the membership function used, different kinds of zones can been identified, such as: *enumerative*, *premetric declarative*, *polygonal*, and *pure functional*. Figure 4.5 represents the relationship between the concepts of zone, space, and location.

A zone characterised by an *enumerative membership* function (also defined as an *enumerative zone*) has a non-empty set of characteristic locations: the membership function is based on the standard belonging relationship defined in set theory and all the locations belonging to the zone are identified through the enumeration of the set of characteristic locations. As an example, a specific area of a grid space can be represented with an enumerative zone by listing all the cells included in such area. On the other hand, a *polygonal membership function* is related to a *polygonal zone* in which the characteristic locations are the vertexes of a polygon, and indicates the inclusion of a location in such polygon. As an example, the zone representing a specific room within a two dimension space that depicts a floor of a building can be obtained by a polygonal membership function with the vertexes of the polygon as the zone characteristics locations.

A *premetric declarative zone*, which is a zone that has a *premetric mem-*

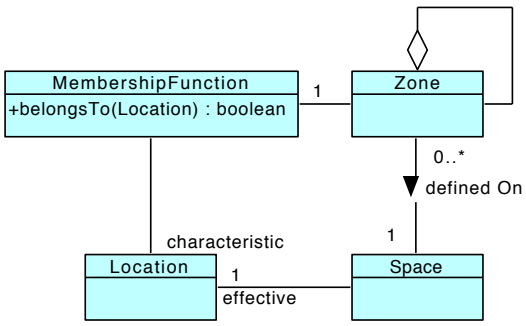


Figure 4.5: Space, Location, and Zone.

bership function, only features a single characteristic location. Its membership function thus includes all and only the locations situated at a given distance from the characteristic locations. A clear example of this kind of zone can be the representation of the detection area of a RFID reader in a Cartesian space.

Finally, a *pure functional zone* has a *functional membership* function that uses mathematical expressions defined in terms of the space coordinate system. An example can be the following one: for each pair of locations (x, y) in a space S and for any given location (x_0, y_0) and (x_1, y_1),

$$f(x) = \begin{cases} true, & \text{if } x_0 < x < x_1 and y_0 < y < y_1 \\ false, & \text{otherwise} \end{cases}$$

This example defines a rectangular zone, in which all locations between (x_0, y_0) and (x_1, y_1) are included.

Figure 4.5 also pictures that a zone can be seen as an aggregation of other zones. This allows to create a zone that contains *sub-zones*, which can be useful in many cases, as an example, when dealing with continuous spaces and locations defined by real values, a simple enumeration can be an issue: it is impossible to get a positive response when using an enumeration of numbers with infinite precision. Using the concept of sub-zones, the enumeration of real numbers can be expressed as an enumeration of zones, where each zone features a premetric declarative membership function with each real value as characteristic location and a distance ϵ as small as needed that defines the precision of acceptance.

As pictured in Figure 4.6, zones and membership functions are build respectively from the meta level descriptors *ZoneModel* and *MembershipFunctionModel* as the rest of the spatial models.

4.2.4 The Stimulus

Usually, data coming from physical or software sensors, is strictly related to the sensor itself. This means that without the knowledge of the characteristic of the source it is difficult, if not impossible, to understand and manipulate such

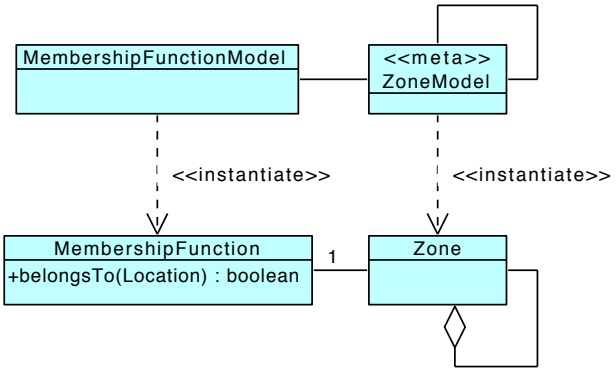


Figure 4.6: The Zone Model.

data. One of the main contribution of this work deals with the representation of data from sensors that have been completely dissociated from the sensing devices exploiting the concepts of spaces, zones, and locations introduced in Section 4.2.1. Figure 4.7 represents the general structure of a *Stimulus*.

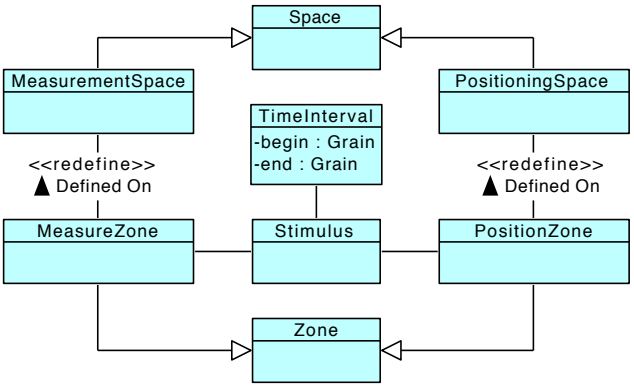


Figure 4.7: The Stimulus.

A stimulus is defined as any information related to a physical event and it is composed by three main information:

- *Time Interval*, the acquisition time as defined in Chapter 3.
- *Measure Zone*, the reading payload.

- *Position Zone*, the reading location.

Measure Zone

As pictured in Figure 4.7, the measure zone is a specialisation of zone and it is referred to a *Measurement Space*, which is a specific kind of space devoted to represent all the possible values that an information source can produce. Within the measurement space are represented all the characteristics of the allowable locations. As an example, Figure 4.8 pictures the details of a possible representation of the *Temperature Space* subjective to the thermometer *therm1*. In this example, the space features a single dimension, which is the dimension of temperature values, the precision is referred to the precision that values are expressible with and their unit of measure is Celsius. The dimension also has its boundaries specified as two temperature values, -10 and 40 that represents respectively the lowest and highest temperatures sensible by the sensor. From this example it is clear that all the information from sensor `therm1` that is related to the values it can provide, are embedded within the specification of its measurement space.

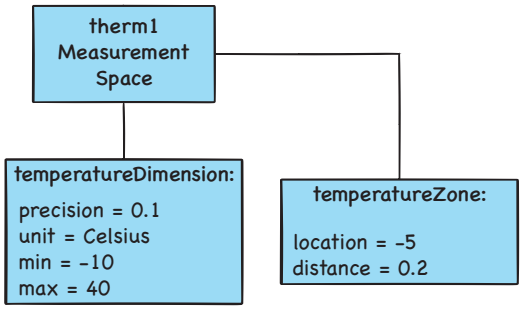


Figure 4.8: Therm1 Temperature space and zone example.

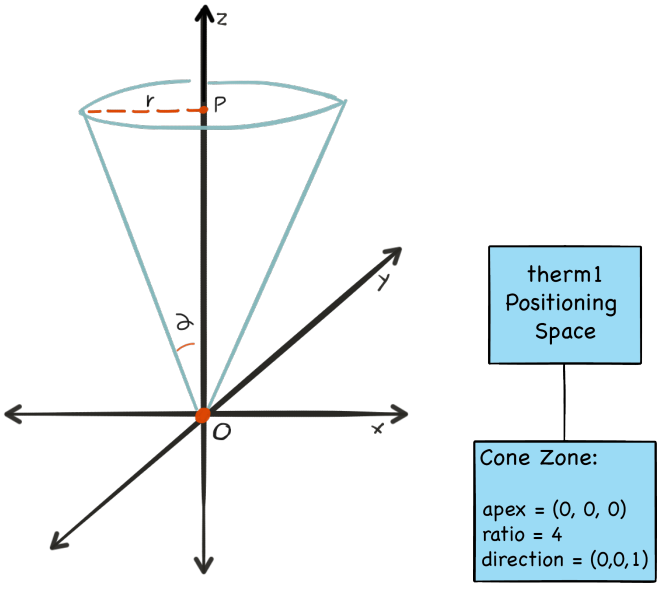
The definition of a zone on a measurement space can be seen as expressing the payload of a single reading. At first, the use of a zone instead of a single location in order to represent the payload of a stimulus may seem an over-complication, but after a more careful examination this choice allows the representation of more meaningful information. As an example, a simple temperature probe such as the one in the thermometer defined in the example scenario can be considered. It is composed by a metal component that is physically designed to output a voltage signal that is linearly proportional to the local temperature. The most intuitive approach would be to use a temperature value representing the conversion from the voltage signal to the corresponding Celsius value. However, the precision of sensors may differ with respect of the values read. As an example, the data sheet of `therm1` could reasonably state something like:

$$Accuracy = \begin{cases} \pm 0.2^\circ\ C, & \text{from } -10^\circ\ C \text{ to } 0^\circ C \\ \pm 0.1^\circ\ C, & \text{from } 0^\circ\ C \text{ to } 30^\circ C \\ \pm 0.2^\circ\ C, & \text{from } 30^\circ\ C \text{ to } 40^\circ C \end{cases}$$

Using a location to represent the current reading, such information would be needed at any level, in order to correctly interpret and use the temperature value. On the other hand, by using a premetric declarative zone, the accuracy of each reading could be embedded within the measure itself: so for a value between $-10^\circ\ C$ and $0^\circ\ C$, the value of the ϵ parameter would be 0.2 as it is in the example in Figure 4.8. Similarly in the context of `cam1`, the measurement zone would be an enumeration of triplets, defined by one value for each dimension R, G, and B pictured in Figure 4.4b.

Position Zone

The position zone represents where the measure zone is placed. A simple thermometer like `therm1`, does not provide (or know) any information about where its readings are positioned: in this scenario, the position zone, would be irrelevant and usually corresponds to the origins of the position space. A more interesting scenario, is the one where the sensor is a different kind of thermometer, like an infrared thermometer, which infers the temperature from the thermal radiation emitted by the object toward it is pointed at. In this case,



(a) Cone representation (b) Cone model

Figure 4.9: The cone model and representation.

the position zone is represented by a cone as in Figure 4.9. Similarly, a measure zone for `cam1` would be represented by an enumeration of locations (pixels) that belongs to its measurement space and represents the acquired image.

Figure 4.9b pictures a cone in a three dimension Cartesian space, the apex of the cone is placed on the origin of the axes and it opens toward the positive Z axis. The opening angle α determines the radius r at the height z_p. In sensors it is usually expressed as a *ratio* between height of the cone section and depth of the base, also known as distance to spot ratio. Infrared thermometers usually do not provide distance measures, which is why the model in Figure 4.9b does not includes the point P: the cone is supposed to extend itself indefinitely, but it consider the apex of the cone in $(0, 0, 0)$ and the direction represented by the vector $(0, 0, 1)$ (it opens toward the positive Z axis).

Similarly, the information produced by `cam1` should be positioned in the area of a space that falls in field of view of the camera, as sketched in picture Figure 4.10.

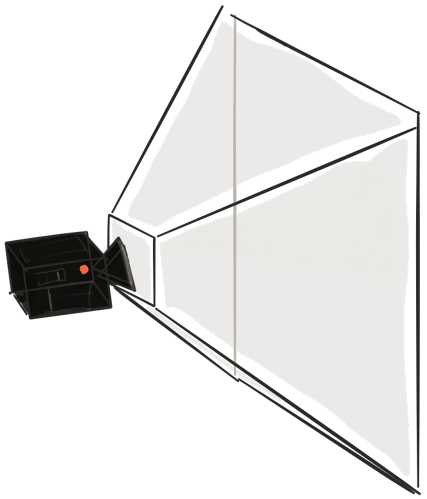


Figure 4.10: Camera positioning space.

It is worth noting that the distinction between positioning and measurement concepts is purely conceptual: they are all spaces and zones, as defined in Section 4.2.1.

4.2.5 The Source

As mentioned in Section 1.3, one of the main issue of currently available proposals is the fact that the knowledge of low level physical devices is spread at each level of the system, up to the applications. In this work the more general concept of *Source* is introduced. A source is intended as a role, instead of a physical object. For example, when considering the thermometer `therm1`, it

acts as a source with respect to the component that manages the temperature in room `room1` (`room1Temp`); at the same time, `room1Temp` could act as a source to another component that regulates the temperature in the floor.

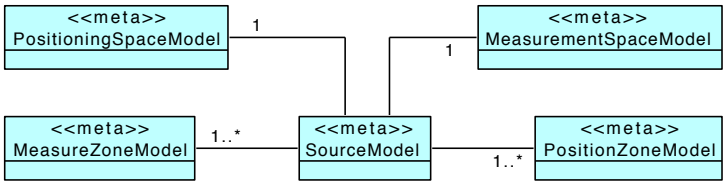


Figure 4.11: The Source Model.

As pictured in Figure 4.11 a *SourceModel*, from which a source is instantiated can be fully defined by:

- A *MeasurementSpaceModel*, which defines the characteristics of the measurements that the source provides.
- One or more *MeasureZoneModel*, which states how the measurements belonging to the measurement space are expressed.
- A *PositioningSpaceModel*, representing the positioning space in which those measurements are contextualised.
- One or more *PositionZoneModel*, which expresses how measurements are positioned inside the positioning space.

A source can feature more than one measure and position zone model, this allows for more complex sources that could represent their data in different manners.

4.2.6 The Mapping Function

Given two different spaces, the concept of *Mapping* relates one zone defined on one space with another zone defined on the other one. Starting from [91] three kind of mappings can be defined:

- Explicit Mappings.
- Projective Mapping.
- Implicit Mappings.

An explicit mapping is an ordered pair of zones defined of different spaces, possibly build from different models: given the spaces S_1 and S_2, with $S_1 \neq S_2$, the ordered pair (Z_{S_1}, Z_{S_2}) is an explicit mapping between the zones $(Z_{S_1}) \subseteq S_1$ (the source) and $(Z_{S_2}) \subseteq S_2$ (the target). It is important to note that the target

zone may be defined independently from the source zone. If, on the other hand, the target zone is the result of the application of a projection to the source zone, the mapping is a projective mapping, and it is described by the types of the involved spaces and the types of the respective zones. Finally, defined SM the set of all the defined mappings and Z_a and Z_b defined as two zones referred to different spaces, an implicit mapping (Z_a, Z_b) is derived if there exist n zones $Z_1, ..., Z_n$ such that $(Z_a, Z_1), (Z_1, Z_2), ..., (Z_n, Z_b) \in SM$ for $n \geq 1$.

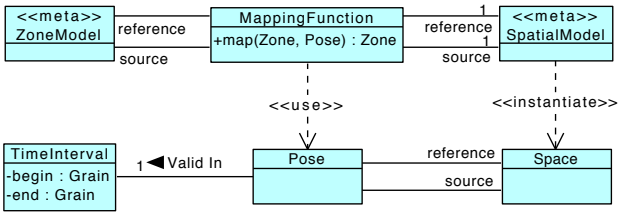


Figure 4.12: The Mapping Function.

As mentioned, an explicit mapping is used to relate zones that are independent, projective mappings functions (mapping functions from now on) are thus the best choice, because the zones they produce can be seen as the representation of the source zone in the target space. In this work, mapping functions are used as connecting components between abstraction layers: for example, they can be used to abstract stimuli coming from physical sensors that are contextualised in subjective spaces, into other stimuli, referred to spaces that are subjective with respect to the software component interested in the readings. This pattern can be repeated many times as required by the abstractions needed.

For example, the stimuli produced by the simple thermometer `therm1` represents the temperature in a specific position of the sensor subjective space. By applying a projection mapping to the position zone of the `therm1` it is possible to obtain a new position, that represents the same information within the position space of `room1`.

With the aim of using a mapping function in order to position a stimuli (for example, from a sensor subjective space, to a space representing a room), the real *position* of the sensor itself inside the room comes in place. Since the aim of this work is to move from physical devices toward the concept of spaces, the position of a generic source with respect to a destination space is expressed as the *Pose* of the source position space with respect to its reference (target) space. For example Figure 4.13 gives a graphical representation of the concept of pose for `therm1` positioning space within the `room1` positioning space, it also gives a glimpse of how the conical zone defined in Figure 4.9a needs to be represented with respect of the `room1` perspective. The pose of a space with respect to another space is strictly dependent on the types of the two spaces: for example, when dealing with Cartesian spaces, the most common information needed to define a pose are:

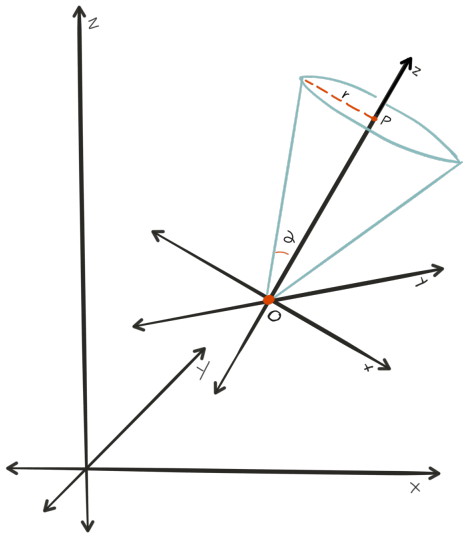


Figure 4.13: The Pose of the `therm1` space.

- a rotation and translation matrix
- a set of multipliers, in order to address the differences in scales between the two spaces. It may be one for each dimension, or a single one for all of them.

Figure 4.14 shows the mapping of the cone zone from the subjective space of `therm1` to the equivalent zone in the `room1` space. The conversion of the apex location its base on the roto-translation matrixes (they are not reported here, but can be calculated by different tools already available). The ratio of the destination zone depends on a interpolation of the scale values and the direction, finally, the destination direction vector is calculated applying to it the same rotation matrixes. What could also be included in the room perspective is the point P as showed in Figure 4.13 as it is reasonable to delimit the cone to the boundaries of the room, nonetheless this could need a different and more complex polyhedric representation if the cone intersects the corners of the space.

While it is more common to think about the concepts of pose and mapping with positioning spaces, the same paradigm can be applied to measurement spaces. Referring to the temperature measurements considered until now, it is possible to define, as an example, the mapping between the measure contextualised in the subjective space of `therm1`, in a measure contextualised in a different temperature space, like the one referred to `room1`. Considering that `therm1`'s temperature space is expressed in Celsius degrees, while the `room1`'s is in Fahrenheit degrees, the mapping function can be seen as the conversion function, while the pose are the parameters that `align` the Celsius scale with respect to the Fahrenheit scale.

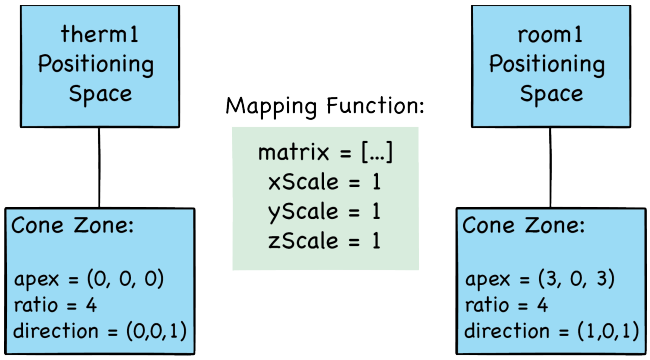


Figure 4.14: The mapping of a cone.

By putting together both a positioning and a measurement mapping function, it is then possible to completely contextualise a stimulus in a different space. Figure 4.15 shows how abstraction of information works under the presented paradigm.

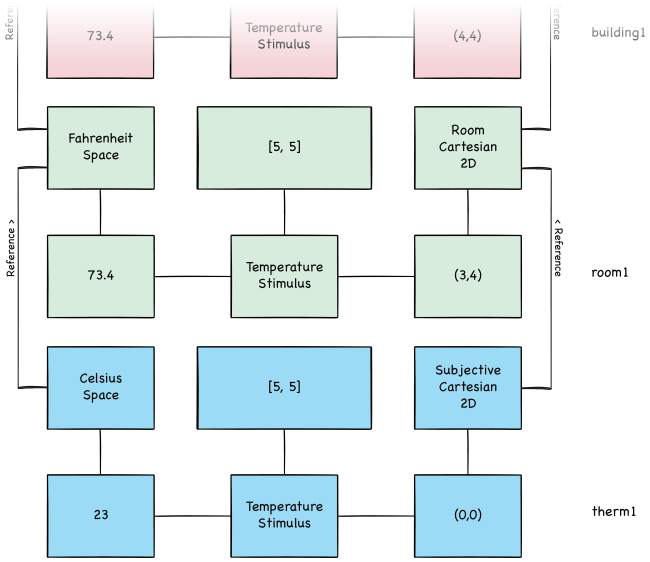


Figure 4.15: The stimulus mapping chain.

4.3 SPACES Concrete Architecture

This Section covers some of the details of the concrete design of the SPACES concepts. The model is based on a Desktop environment where information can

be received from sensors (be they concrete or emulations). The possibility to exploit virtual sensors allow simulations with unavailable sensors and the ability to reproduce the feeding of data from data acquired offline.

4.3.1 Space and Location

The `Space` and `Location` classes as defined are pictured in Figure 4.16. As pictured, a space is characterised by the type of locations that can relate to it. The concept of premetric is not considered in this implementation, but the definition `MembershipFunction` previously introduced for zones, has been applied also to spaces.

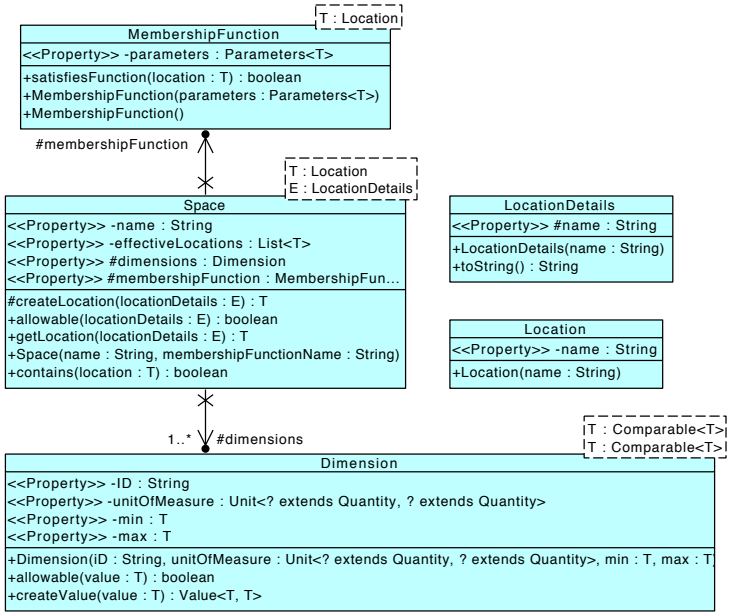


Figure 4.16: The `Space` and `Location` classes.

For consistency purposes, spaces are in charge of creating locations following the factory method pattern; the `LocationDetails` class has been introduced as container of information needed in order to produce a location in a space. Finally it is also noteworthy the `allowable` method that checks if the details used to create a location are consistent with the space type. A number of spaces have been defined as examples and for testing purposes, they are pictured in Figure 4.17.

- `NamesSpace` represents a simple spaces composed by a single dimension of string values, it can be used, as an example, for representing an enumeration of names.

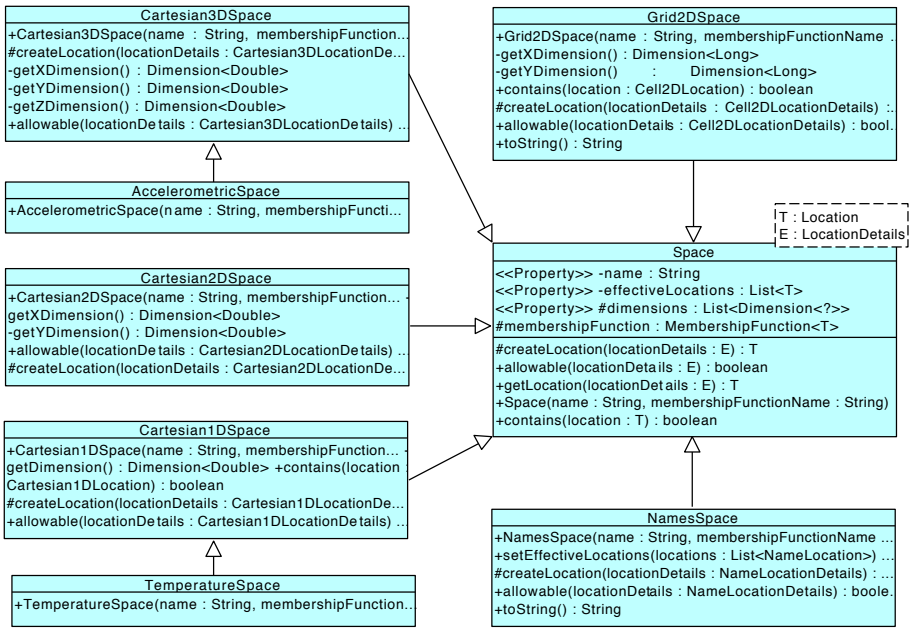


Figure 4.17: The implemented `Space` specialisations.

- `Grid2DSpace` is two dimensional discrete space, it can be used for specific kind of measurements or for simple 2D positions.

- `Cartesian1DSpace`, `Cartesian2DSpace`, and `Cartesian3DSpace` are the most versatile spaces, they are respectively composed by one, two, or three dimensions of double values and can be used both for contextualise measures and positions.

- `TemperatureSpace` and `AccelerometricSpace` are examples of specific cartesian spaces, with one and three dimensions respectively. They have been used during the testing of the framework.

In a consistent way, locations have been specialised too, Figure 4.18 shows the implemented localisations.

The `NameLocation` class contains a string value and represents a specific version of the `OneValueLocation` that is able to be instantiated with a generic `Value` type `T`. The `TemperatureLocation`, as another example, constraints `T` to be a double value, making it suitable to represent temperature readings. Similar locations with two and three values have been defined as the `TwoValuesLocation` and `ThreeValuesLocation` classes, which parametrisation allow to obtain specific cartesian locations such as the `Cartesian3DLocation` class or the `Cell2DLocation` for two dimensional grids spaces. It is noteworthy how, being spaces parametrised with location types, it is easily possible to build, as

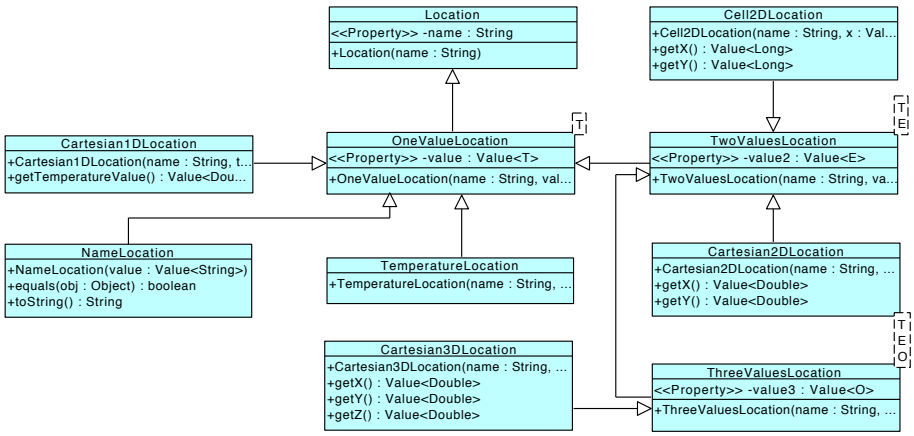


Figure 4.18: The `Location` specialisations.

an example, a two dimensional cartesian space that produces only locations of the type `Cartesian2DLocation` in order to guarantee type consistency.

4.3.2 Dimension and Value

In Figure 4.16 the `dimensions` parameter for the space class is presented. Figure 4.19 pictures the `Dimension` and `Value` classes as introduced in Section 4.2.2. A dimension is characterised by its boundaries expressed as `min` and `max` and a `unitOfMeasure`. Similarly to locations and spaces, `Values` are created by the dimension class that acts as factory and an `allowable` method is used for consistency. The type of values a dimension can create is bounded to the dimension parametrisation. A value is specific to a single dimension and retains its `ID`.

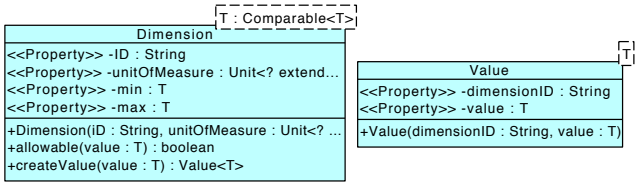


Figure 4.19: The `Dimension` class.

Being parametric, nor specific dimensions or values had to be defined.

4.3.3 Zone

The `Zone` class is shown in Figure 4.20. Every zone is characterised by a reference space and a set of parameters that depends on the zone type, characteristic

locations of a zone falls within the parameters. As definition every zone employs a `MembershipFunction`, in order to offer the `contains` operation and determine if a given location belongs to the zone.

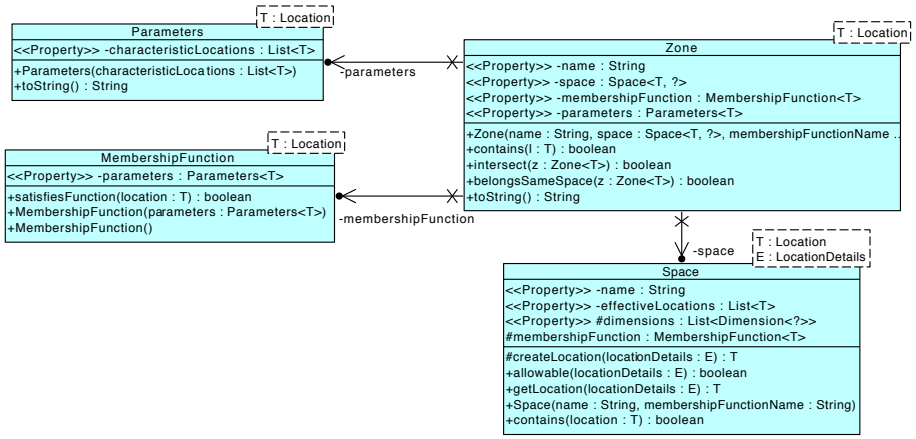


Figure 4.20: The `Zone` and `MembershipFunction` classes.

Being a zone fully described by its membership function, specifications of the latter are enough in order to obtain differentiated versions of the former. Figure 4.21 pictures some of the developed functions.

- `EnumerationMembershipFunction` is the simplest form of function defined: it checks whether or not a given location is part of the collection of characteristics locations.
- `OneDimensionMembershipFunction` is defined to check if a given one dimension location value falls within a single distance `distanceValue`. it can be used as an example to define a zone represented by a single real number and a small threshold of tolerance, as mentioned in 4.2.3.
- `SingleDistance` and `TwoDimensionDistance` functions are related to two dimensional spaces and zones. TwoDimensionDistance allows to define a distance for each of the two dimensions of the space and thus obtain an ellipse of acceptance. The single distance variation, while not actually a specialisation can be seen as a specific instance of the two distance case with equal values: the result is a circle of acceptance around the centre value, which is the characteristic location of the zone.
- `PoligonalMembershipFunction` builds a polygon of an arbitrary number of edges defined by the characteristics locations of the zone that act as vertexes. The `Polygon.contains` function is then used to check if the location passed to the `satisfiesFunction` function are part of the defined polygon or not.

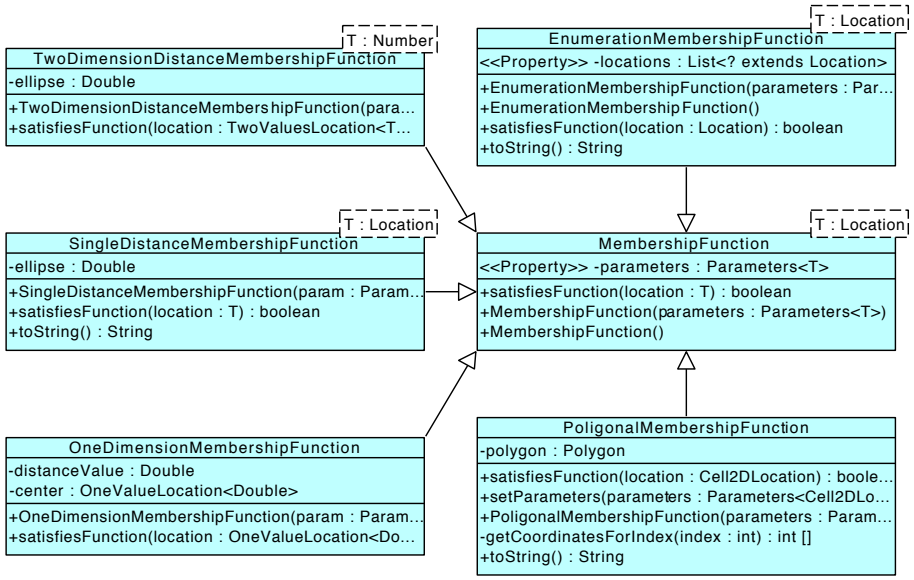


Figure 4.21: The `MembershipFunction` specialisations.

4.3.4 Stimulus and Measure

SPACES is aimed at normalising and contextualising sensor information into stimuli, as introduced in Section 4.2.4. however in order to do so a number of different steps have been considered.

Figure 4.22 pictures the `Stimulus` class and how it is composed. The general `Measure` class is defined by a measure zone that holds the information payload and a `TimeInterval` of validity, defined following the principles of TAM and the TDSH. The concept of `SensorMeasure` associates a `Sensor` to the measure and can be used to represent normal sensors data before adding the spatial contextualisation characteristics to the SPACES approach instead of the physical source reference. Finally the Stimulus adds to the measure the positioning information by enclosing a second zone for space contextualisation.

Figure 4.23 shows the mentioned `Sensor` class. It is characterised by a zone that, if available, represents the actual position of the sensor inside the reference positioning space. It also holds a reference to the measurement space that it uses to produce sensor measures from raw data. The more general Source class has both a positioning space and a measurement space and is able act as a factory to produce stimuli, instead of measures.

In order to better understand how raw data is abstracted into measures and stimuli, Figure 4.24 shows the pipeline of a stimulus. The physical sensor produces the raw data, which is only composed by the data itself and a timestamp of acquisition. The SPACES representation of Sensor uses this data in order to contextualise the data within a measure zone and thus outputs a sensor measure,

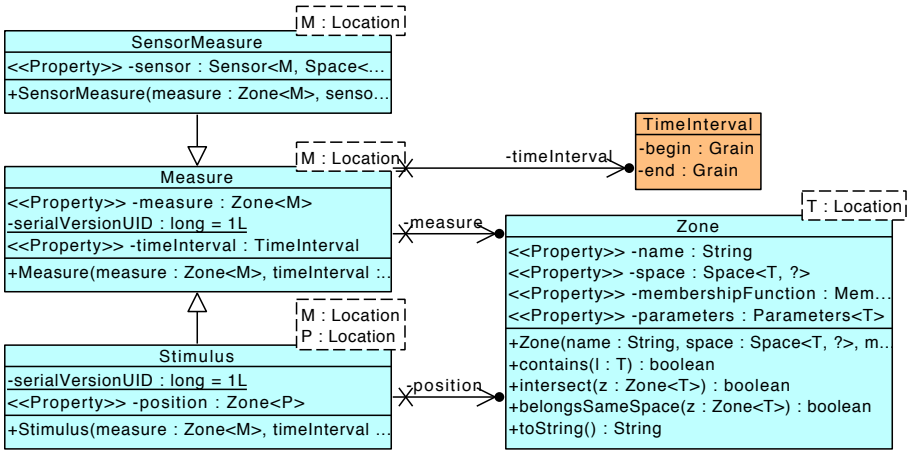


Figure 4.22: The `Stimulus`, `Measure`, and `SensorMeasure` classes.

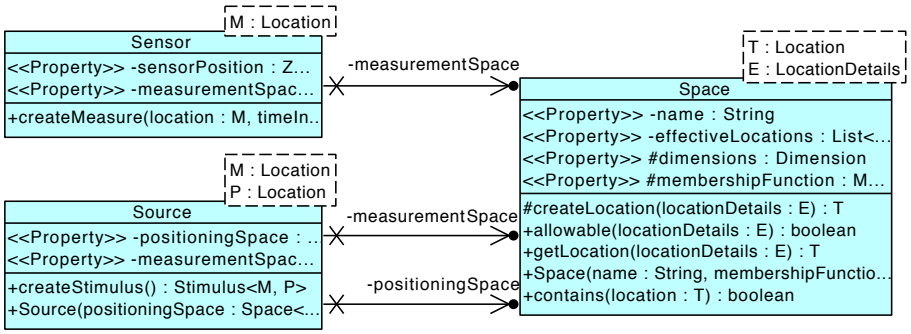


Figure 4.23: The `Sensor` and `Source` classes.

which holds a reference to the sensor. Finally the figure of a source is introduced in order to enrich the the sensor measure and produce a stimulus with a physical position zone instead of the reference to the sensor. It is noteworthy that the abstractions from raw data to stimuli have been proposed as a single operation in Section 4.2.4 as they together represent the change of perspective, from physical sensors, to abstracted and contextualised information, but have been divided into two steps as the two spatial contextualisations are disjoint.

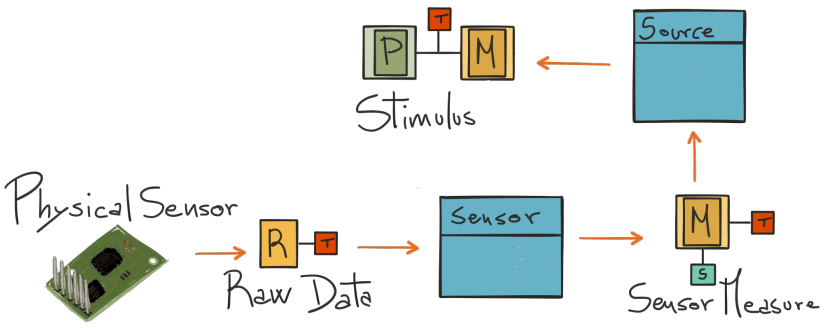


Figure 4.24: The `Stimulus` Pipeline.

4.3.5 Mapping Function

The concrete design of the concept of `MappingFunction` is presented in Figure 4.25. The function is defined by the type of locations (and thus, spaces) that can relate. For example, the `map` function is bound to take as input a zone parametrised with the location type of its source space (`S` in the class diagram) and it produces a zone characterised by the location type `R`, which is the type of its reference space locations. It is noteworthy how only a reference to the *reference* space its needed, as the *source* space is implicitly passed with the `sourceZone` parameter of the `map` function.

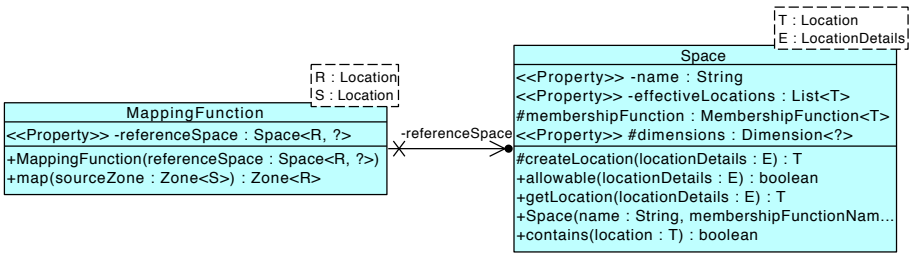


Figure 4.25: The `MappingFunction` class.

4.4 Implementation

This section covers the implementation of the SPACES model. The library is build using the Java language and the Eclipse IDE. The actual implementation is generally consistent with the concrete model presented in Section 4.3 with minor modifications made to fulfil practical aspects that did not appear in the design phase.

For example, the concept of `Orientation` and `OrientedZone` have been added to model those zones that may have a specific orientation within their

space. the `Parameter` class is used to define both the parameters needed to instantiate a `Zone` and a `MembershipFunction`.

4.4.1 Implementation Choices

In the developing of the library, some external resources have been exploited. For example, in order to model units of measures within the `Dimension` class, the `JScience`[1] library is exploited. `JScience` is a freely available library that provides implementation of units of measurement services along with other mathematical commodities. Another example are the `SingleDistance` and `TwoDimensionDistance` membership functions presented in Section 4.3.3. Both classes exploit the `Ellipse` class of the `java.awt.geom` package. Similarly the `PoligonalMembershipFunction` makes use of the `java.awt.Polygon` package.

4.4.2 The SPACES Packages

The implementation of the SPACES model has been divided into three main packages:

1. `SpaceCore`.
2. `DataCore`.
3. `Library`.

The `spaceCore` package is pictured in Figure 4.26 and contains the reifications of all the core concepts for spatial representation and spatial mappings.

Figure 4.27 pictures the `dataCore` package, where are implemented the concepts related to stimuli and measures, as well as the `Sensor` and `Source` classes.

Finally, Figure 4.28 shows the `library` package. This package has been added in order to give developers some basic instruments to deploy their software components exploiting SPACES. The `library` package contains all the specialisations of the core classes found in the `spaceCore` and `dataCore` packages.

[1] http://jscience.org

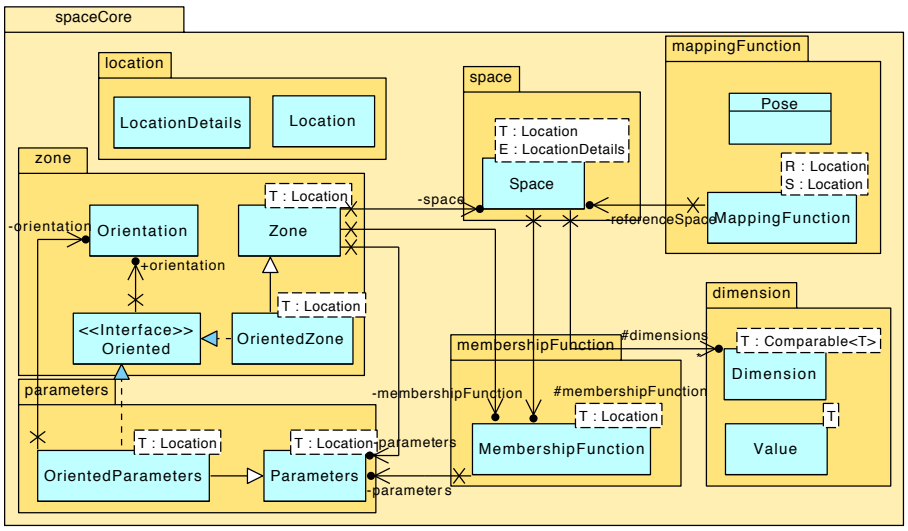


Figure 4.26: The `spaceCore` package.

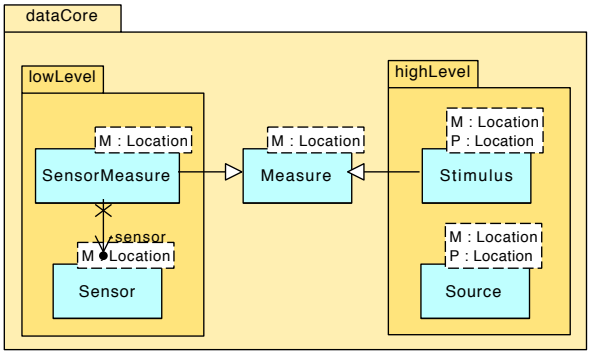


Figure 4.27: The `dataCore` package.

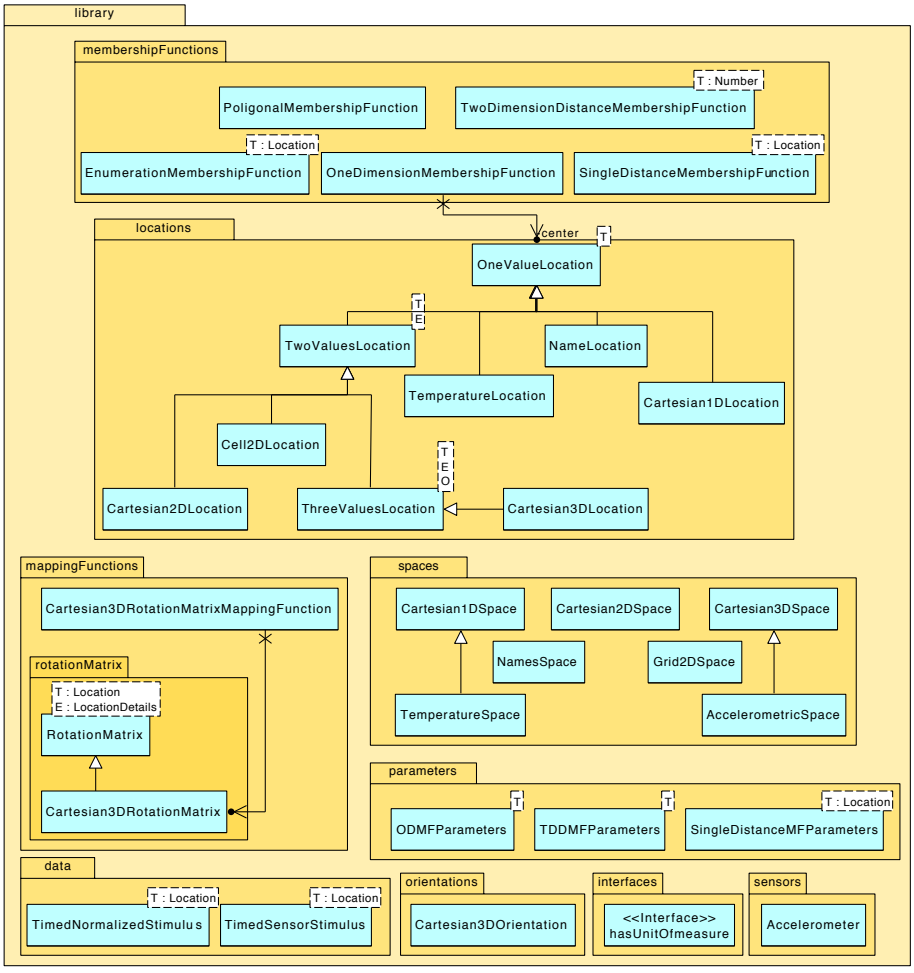


Figure 4.28: The `Library` package.

Sub-packages have been defined to contain the various concepts specialisations. For example the `library.locations` package contains all the various locations needed to represent the locations of the corresponding spaces implemented in the `library.spaces` package. As pictured in Figure 4.29, the final specialisations removes the template parameters, forcing the various locations to assume the right kind `Values`, such as `String` for the `NameLocation` class or `Long` for `Cell2DLocation`.

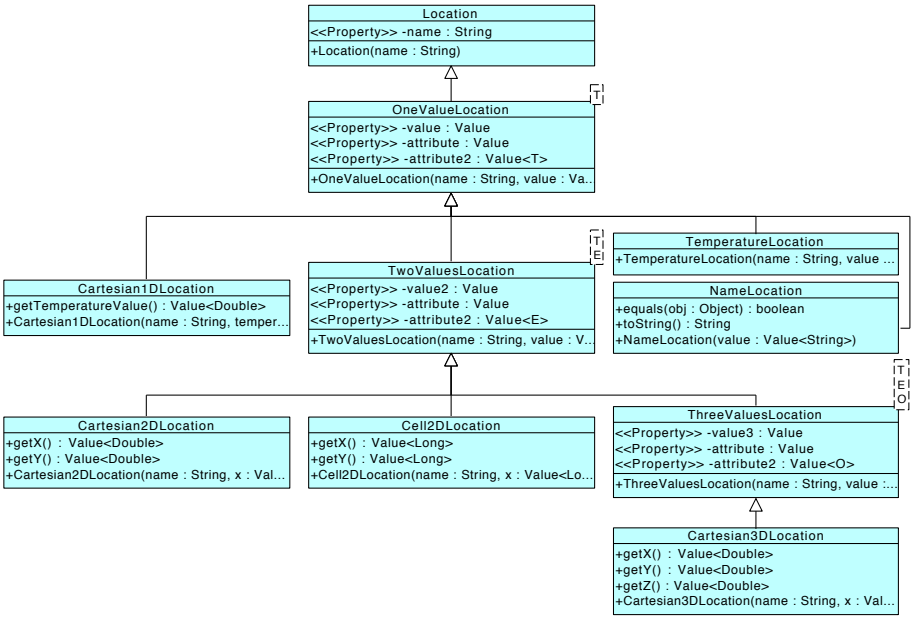


Figure 4.29: The `library.locations` package.

Figure 4.30 shows the available mapping functions. Defining the correct mapping function for specific zones in given spaces is a very hard and space-dependent matter. The current implementation features some basic mappings such as name to name from enumeration space to enumeration space and name to zone for enumeration space to any other space. More interesting mapping functions are mathematical based.

For example, Figure 4.30 pictures the `Cartesian3DRotationMatrixMappingFunction`, which maps zones from a three dimensional cartesian space into zones of another space of the same type. The `map` function is based on its `Cartesian3DRotationMatrix`, that reifies the concept of pose of a space with respect to another space introduced in 4.2.6. The result is a zone of the same shape (proportions may change) and type of the original one, but contextualised in the reference space.

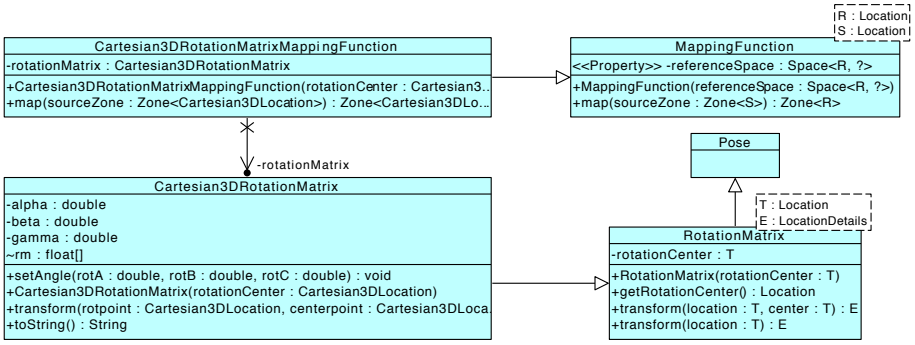


Figure 4.30: The `library.mappingFunctions` package.

4.5 Normalisation Case Study

In this section the case study presented in 2.2.2 is considered from the SPACES perspective. In order to achieve a spatial representation of the data that is not dependent on the physical source, it is necessary to:

1. Correctly identify how the data has to be represented from the point of view of the sensor.
2. Define how the normalised data should be represented.
3. Identify the correct mappings between the two representations.

The rest of this section will cover this points both from a measurement and positioning point of view, for each of the data type in use (accelerations and audio).

4.5.1 The Concepts Needed

In order to represent accelerations and microphonic data exploiting the SPACES approach, a number of concepts have to be defined. Specifically, each type of data needs to be specified in terms of measurement space and positioning space.

Acceleration

Measurement To represent acceleration measures, the following concepts are needed:

1. An acceleration measurement space related to the accelerometer. Such space has to be able to represent the inner characteristics of the sensor in use. For example its allowable values should be bounded to the minimum and maximum values the sensor can sample (-8; +8) and its unit of measure should be g, the gravity unit of measure, where $1g = 9.81 m/s^2$.

2. A corresponding measurement space to represent accelerations at room level, which may not be completely consistent with the sensor related one. As an example its unit of measure should follow the International System of Units and be directly in m/s^2.

3. A mapping function able to transform the stimuli contextualised in the former space into corresponding stimuli contextualised in the latter. In this case, it consists in a roto-translation of the values to compensate for change of perspective and a transformation corresponding the change of scale. As the acceleration is a directional data, the position and orientation of the sensor within the room is used for the roto-translation, the assumption of its availability is mentioned in 2.2.2.

Positioning Accelerometers do not provide any information about the position of their readings, the positioning spaces are then as follows:

1. The sensor positioning space is irrelevant as no information about actual positions is provided. For better consistency it is defined as a three dimensional cartesian space. The position of the data is defined as the origin of the space.

2. The room positioning space is a three dimensional cartesian space, in which every information interpreted at room level is positioned.

3. The mapping function that relates the two spaces is in theory the same roto-translation function defined for the acceleration measurements, but is in fact more trivial: being the position of the sensor available in room coordinates, and being every acceleration positioned contextually with the sensor, it is sufficient to use the sensor position to contextualise accelerations from the positioning perspective of the room.

Audio

In this context the linear microphonic array is composed by the two microphones distinctively and their data is received disjoint.

Measurement The concepts required for representing the sound information from the microphones are as follows:

1. An audio space, related to each microphone, with a single dimension related to the intensity of the sound (the signals used are not directly in decibel, the standard audio unit of measure). Since the acquisition from the microphones is performed in chunks (in accordance to 3.6.2), the locations of this space should not be single measures, but each chunk.

2. A corresponding space for measuring the audio from the room perspective. Being the room equipped with two identical microphones, it is reasonable to keep the same data representation at room level.

3. As no transformation of measurements is due, there is no need of an actual measurement mapping function.

Positioning The positioning information of audio data is defined as follows:

1. A three dimensional cartesian space for positioning the audio chunks from the sensor perspective. Microphones are characterised by an angle of acquisition, moreover a maximum distance of perception can be empirically defined in the given context. These information have a practical impact on the positioning of audio information. Specifically, the zones in which stimuli are to be contextualised will have to embed these intrinsic characteristics of the sensors. The result zone is therefore similar to the conical zone model presented in Section 4.2.4 and sketched in Figure 4.9.
2. The positioning space of the room in which audio is contextualised is in fact the same cartesian space defined for positioning accelerations.
3. As the two spaces are again both three dimensional cartesian spaces, the already defined mapping function is to be used. Specifically the rotation is to be applied to the characteristics locations that define the conical zone of the microphones.

4.5.2 Implemented Classes

Here the actual classes representing the concepts introduced in Section 4.5.1 are presented.

Acceleration

Measurement The concrete measurement classes for representing accelerations are the followings:

1. The sensor space `AccelerationSpace` is a `Cartesian3DSpace`, with the three dimensions holding `double` values, with g as `unitOfMeasure`, and their boundaries set to -8 (min) and $+8$.
2. The `AccelerationSpace` at room level is identical, except for the fact that it holds values in m/s^2.
3. The mapping function used to map sensor accelerations into room accelerations is a `Cartesian3DRotationMatrixMappingFunction` and its core is represented by its `map` method shown in 4.1.

Listing 4.1: The `Cartesian3DRotationMatrixMappingFunction` `map` method.

```
1 @Override
2 public Zone<Cartesian3DLocation> map(Zone<Cartesian3DLocation>
      ↪ sourceZone) {
3
```

```
4       Cartesian3DOrientation c3o = (Cartesian3DOrientation) ((
            ↪ OrientedZone<Cartesian3DLocation>) sourceZone).
            ↪ getOrientation();
5           rotationMatrix.setAngle(c3o.getaAngle(), c3o.getbAngle
                ↪ (), c3o.getcAngle());
6
7       Parameters<Cartesian3DLocation> parameters = new Parameters
            ↪ <Cartesian3DLocation>(new ArrayList<
            ↪ Cartesian3DLocation>());
8       for (Cartesian3DLocation location : sourceZone.
            ↪ getParameters().getCharacteristicLocations()){
9           Cartesian3DLocationDetails locationDetails =
                ↪ rotationMatrix.transform(location);
10          Cartesian3DLocation newLocation = ((Cartesian3DSpace)
                ↪ getReferenceSpace()).getLocation(locationDetails)
                ↪ ;
11          parameters.getCharacteristicLocations().add(newLocation
                ↪ );
12      }
13
14      Zone<Cartesian3DLocation> mappedZone = new Zone<
            ↪ Cartesian3DLocation>("mapped" + sourceZone.getName(),
            ↪  getReferenceSpace(), sourceZone.
            ↪ getMembershipFunction().getClass().getCanonicalName()
            ↪ , parameters);
15
16      return mappedZone;
17  }
```

As mentioned, acceleration needs to be represented by an oriented zone, which explains the explicit cast at line 4 in 4.1. The `rotationMatrix` of the function is then updated to take into consideration the updated orientation of the source zone. Then, for each characteristic location of the source zone, corresponding locations are created exploiting the `transform` function of the `rotationMatrix`. Finally a new zone is created in the reference space, with the same membership function type of the source zone (an `EnumerationMembershipFunction` in this example) and the newly obtained locations as parameters.

The `transform` function is shown in Listing 4.2. The new location values `newX`, `newY`, and `newZ` are calculated by applying the rotation matrix `rm` and then normalised to `centerpoint`, the point around which the rotation is carried out. In this case the rotation happens around the origin of the axes $(0, 0, 0)$ so the normalisation has no effect.

Listing 4.2: The `transform` method.

```
1   @Override
2   public Cartesian3DLocationDetails transform(Cartesian3DLocation
        ↪  rotpoint, Cartesian3DLocation centerpoint){
3           double px = rotpoint.getX().getValue() - centerpoint.
                ↪ getX().getValue();
```

```
4          double py = rotpoint.getY().getValue() - centerpoint.
               ↪ getY().getValue();
5          double pz = rotpoint.getZ().getValue() - centerpoint.
               ↪ getZ().getValue();
6
7          double newX,newY,newZ;
8
9          newX = rm[0]*px + rm[1]*py + rm[2]*pz;
10         newY = rm[3]*px + rm[4]*py + rm[5]*pz;
11         newZ = rm[6]*px + rm[7]*py + rm[8]*pz;
12
13         newX += centerpoint.getX().getValue();
14         newY += centerpoint.getY().getValue();
15         newZ += centerpoint.getZ().getValue();
16
17         return new Cartesian3DLocationDetails(rotpoint.getName
               ↪ (), newX, newY, newZ);
18     }
```

The rotation matrix `rm` is obtained as in Equation 4.1, where a, b, and g, represent the α, β and γ parameters set with the `setAngle` function (Listing 4.1, line 5).

$$rm = \begin{bmatrix} \cos a * \cos b & \begin{matrix}(\cos a * \sin b * \sin g) - \\ (\sin a * \cos g)\end{matrix} & \begin{matrix}(\cos a * \sin b * \cos g) + \\ (\sin a * \sin g)\end{matrix} \\ \sin a * \cos b & \begin{matrix}(\sin a * \sin b * \sin g) + \\ (\cos a * \cos g)\end{matrix} & \begin{matrix}(\sin a * \sin b * \cos g) - \\ (\cos a * \sin g)\end{matrix} \\ -\sin b & \cos b * \sin g & \cos b * \cos g \end{bmatrix} \tag{4.1}$$

Positioning

1. At sensor level, as no information is produced about the positioning of the accelerations, they are represented in a zone characterised by an enumerative membership function that holds a single location, the origin of the axes. An example of the accelerometer positioning space and the membership function that represents acceleration positions is sketched in Figure 4.31. Here the `Cartesian3DSpace` is defined with its three `Dimensions` that reify the characteristics of the sensor. The `EnumerationMembership-Function` is characterised by the `origin Location`, that represents the origin of the `accelerationPositioning` space.

2. At room level, positions are again represented by a `Cartesian3DSpace` and the location of the accelerations can be derived from the known position of the sensor.

Audio

Measurement

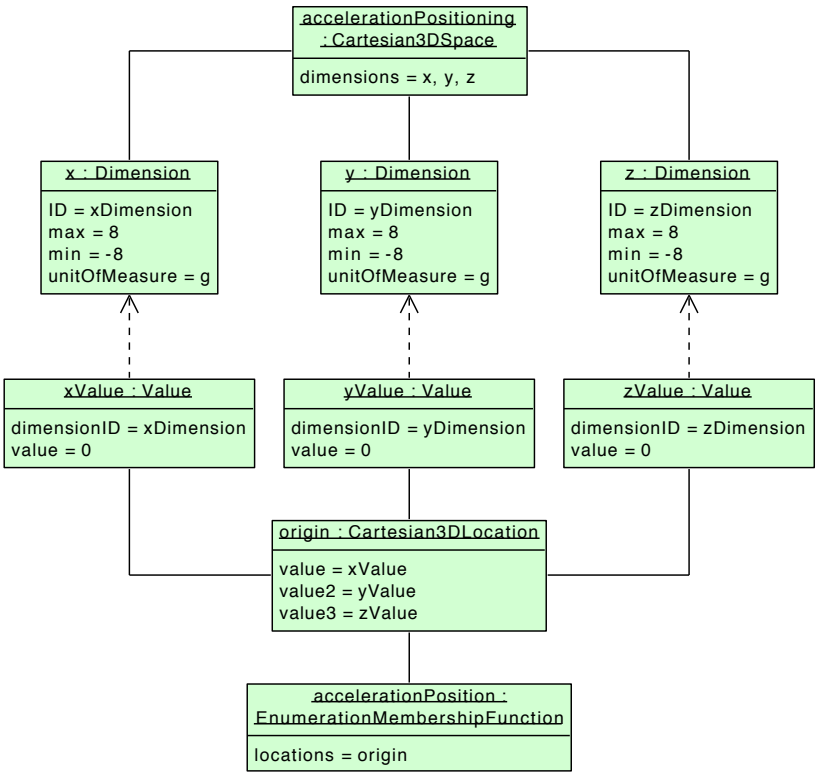


Figure 4.31: Instances representing the accelerometric positioning contextualisation.

1. At sensor level, the classes pictured in Figure 4.32 have been defined: the `AudioSpace` is the measurement space, it has a single dimension and produces locations of type `AudioChunkLocation` by taking as input `AudioChunkDetails`. The concept of audio `Chunk` has been modelled as an ordered list of audio values.

2. The room measurement space for audio information is consistent with the sensor level space.

Positioning

1. The microphone positioning space is a `Cartesian3DSpace`. Audio stimuli are positioned in conical zones defined by the `ConeMembershipFunction` type of membership function, pictured in Figure 4.33. As mentioned in Section 4.2.4, the cone is characterised by:

    - an `apex`, of type `Cartesian3DLocation`.

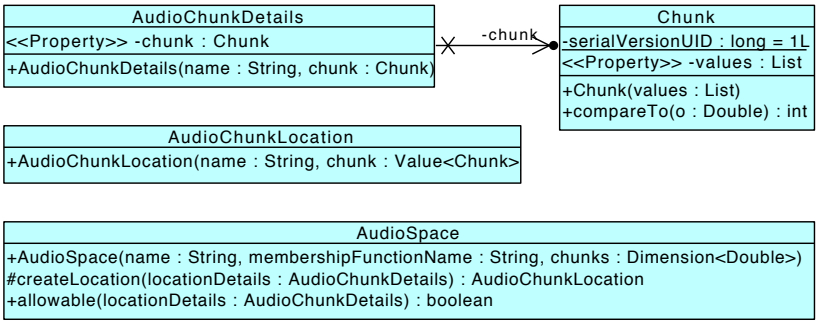


Figure 4.32: The classes for representing audio information.

- an `aperture`, defined as the ratio of which the cone opens, expressed in radians.
- an `infinite` boolean flag, to determine if the cone is unlimited.
- a `centerBasement`, of type `Cartesian3DLocation`; it represent the centre of the cone base if it is not infinite.

2. The room positioning space is the same space in which accelerations are positioned and it has already been introduced.

3. The mapping function used to map the sensor cones into the room space is the already introduced `Cartesian3DRotationMatrixMappingFunction`. In this specific case it is applied to the characteristics locations of the cone (namely `apex` and, if present, `centerBasement`) as these points are sufficient to determine the new cone into the destination space.

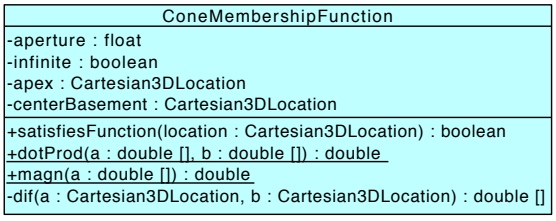


Figure 4.33: the `ConeMembershipFunction` class.

The most interesting piece of code about the positioning of audio information is represented by the `satisfiesFunction` function of the `ConeMembershipFunction` class (Figure 4.33) shown in Listing 4.3. The method calculates `apexToLocVect` that represents the vector pointing to `location` from `apex`, and `axisVect` that is the vector from `apex` to `centerBasement`. The calculation of these vectors is achieved by the `dif` method, listed in Listing 4.4.

Listing 4.3: The `satisfiesFunction` method.

```
@Override
public boolean satisfiesFunction(Cartesian3DLocation location)
    ↪ {

    float halfAperture = aperture/2.f;
    double[] apexToLocVect = dif(apex,location);
    double[] axisVect = dif(apex,centerBasement);

    boolean isInInfiniteCone = dotProd(apexToLocVect,axisVect)
                               /magn(apexToLocVect)/magn(
                                   ↪ axisVect)
                                 >
                               Math.cos(halfAperture);

    if(!isInInfiniteCone) return false;
    if (this.infinite) return true;
        boolean isUnderRoundCap = dotProd(apexToLocVect,
            ↪ axisVect)
                              /magn(axisVect)
                                <
                              magn(axisVect);
    return isUnderRoundCap;
}
```

Listing 4.4: Supporting functions to for the `satisfiesFunction` method.

```
private double[] dif(Cartesian3DLocation a, Cartesian3DLocation
    ↪  b){
    return (new double[]{
        a.getX().getValue() - b.getX().getValue(),
        a.getY().getValue() - b.getY().getValue(),
        a.getY().getValue() - b.getZ().getValue(),
        });
}

private double dotProd(double[] a, double[] b){
    return a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
}

private double magn(double[] a){
    return (Math.sqrt(a[0]*a[0]+a[1]*a[1]+a[2]*a[2]));
}
```

The `satisfiesFunction` logic is that if `location` is lying within the cone if it is lying in its infinite projection. If this condition is met, the function controls the `infinite` parameter of the specific cone, in case the cone is configured as finite, it proceeds to check if `location` falls within the finite section defined by the base (represented by centerBasement). the `location` point is contained

only if the projection of `apexToLocVect`to the axis (`axisVect`) is shorter than the axis itself. The `dotProd` function calculate the dot product between the two locations, while `magn` calculates the magnitude of a point.

4.5.3 The Fall Detection Application

Once the data representation has been defined, the fall detection application can be rethought, the behaviour is as follows:

The basic fall detection component logic remains unchanged: the analysis of the acceleration data is used to infer possible falls. However the characteristics of the sensed data, instead of being part of the application base of knowledge are now fully represented within the acceleration space.

The availability of the spatial contextualisation of the sensed information, enables to exploit the microphonic information only if the acceleration is sensed within the range of the microphones and not consider it if the acceleration happened outside the microphones field.

Furthermore, the application component can be modelled using the SPACES approach: the fall detector can be considered as a source, while the falls it detects as the stimuli it produces.

Figure 4.34 pictures `FallDetector` as a specialisation of `Source` and the `Fall` it produces as a specialisation of `Stimulus`.

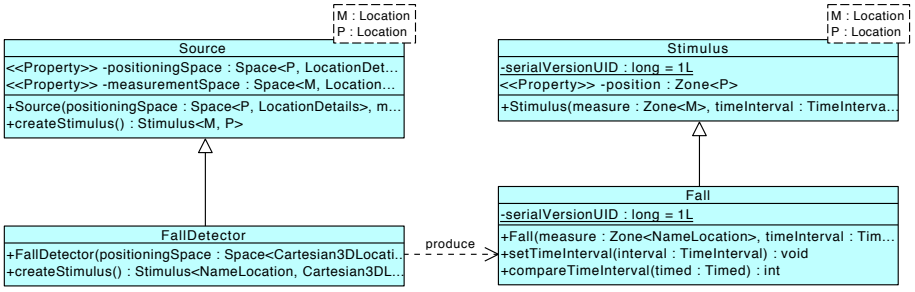


Figure 4.34: The `FallDetector` and `Fall` classes.

The algorithm representing the updated version of the fall detection application is sketched in Figure 4.35 and behave as follows:

- The accelerations produced by the `AccelerationSource` instance are analysed for possible falls.
- If any activity over the predefined threshold is detected the position of the acceleration information, expressed in room coordinates, is considered.
- In case the supposed fall is within the conical zones that contextualise the audio information then the intensity of the audio is checked as in the TDSH case, otherwise the fall is reported directly by producing a `Fall` stimulus. The increased confidence in accelerometric data whenever audio

information is not available (e.g. the data is registered anywhere outside the microphonic array positioning zones), is a conservative choice that favours false positives toward false negatives. The reason for this choice is that, as explained in Section 1.4.1, the consequences for a unreported fall may be severe, while a false negative would presumably result in a unnecessary check from caretakers.

- If the audio analysis does not report any suspicious activity, then the supposed fall is flagged as a false positive, otherwise the fall is confirmed.

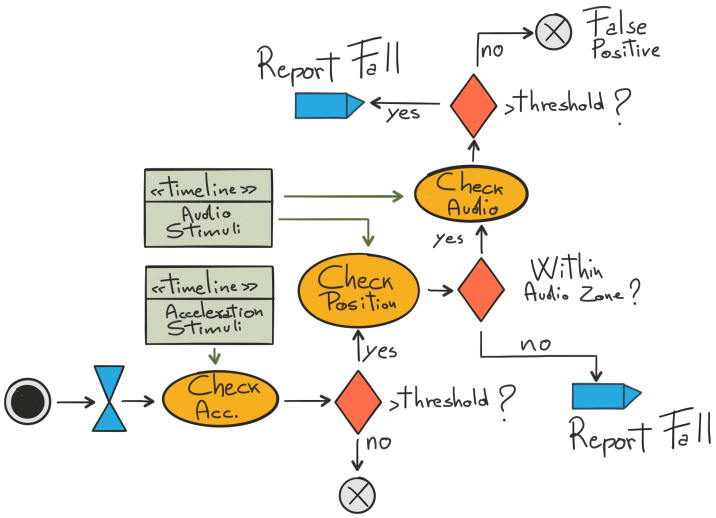


Figure 4.35: The updated application algorithm.

4.5.4 Discussion

The described scenario represents the same situation handled in Section 3.6, but from the middleware perspective and with some hints at the application perspective.

The implementation of the SPACES library highlighted the main challenges related to the spatial representation approach, which are related to the definition of the correct mapping functions for each kind of transformation needed. However these kind of functions are usually based on geometrical transformation and once defined they can be reused in different contexts. As an example the mapping function exploited for contextualising accelerations at room level is the same necessary for mapping the position of any data in a 3D cartesian space, both for measures and positions.

Similarly membership functions may not be trivial to design and develop, but may be re-used as well. As an example the definition of the cone used

for audio information may be exploited to represent the information of a laser temperature probe, such as mentioned in Section 4.2.4.

The Acquisition components have been represented with the *Source* paradigm and the intrinsic characteristics that may be useful to the domain applications have been embedded within their spatial contextualisations. As an example the operating field of microphones described by the conical membership functions.

The final result is that domain applications do not have to meddle with the intrinsic characteristics of the physical devices used to produce the data they rely on, thus making them more resilient in terms of modifications needed in case of hardware change or addition.

Furthermore, the direct spatio-temporal contextualisation of data eases spatial related reasoning in applications such as the conditional checking of audio information presented in this section. Spatio-temporal reasoning represents one of the key aspects in AAL systems, as mentioned in Section 1.2.2.

Finally the SPACES model proved to promote a correct horizontal partitioning and data representation that increase the scalability, reusability, and expandability of the system. As an example, just by instantiating a mapping function that translate falls contextualised in room coordinates into a broaden context, a further component that reasons at building level could deduce the nearest path to the faller in need or what caretaker to notify based on the respective positions of the the caretaker and the detected fall.

CHAPTER 5

Conclusions

This chapter summarises the contributions of this work, the related publications, and the possible future developments.

5.1 Summary of Contribution

This thesis proposed two sets of architectural abstractions that respectively model *data acquisition* and *data presentation*, which can be exploited in any kind of system that deals with data from the filed, such as Ambient Assisted Living systems.

Time driven Sensor Hub (TDSH) are the abstractions related to data acquisition. TDSH accurately separates the mechanisms that realise the dynamics from the configurations that specify the sensors and their acquisition frequencies. Thus, TDSH proposes a model that strongly enforces both reusability and configurability. This allows to easily realise acquisition layers by only selecting by a catalogue the sensors needed, and to configure them in order to acquire date at the desired frequency. If in the catalogue a sensor is not present, it suffices only to program the software component that interface with it.

sPaces Architecture for Contextualising hEterogeneous Sources (SPACES) are the abstractions related to data representation. SPACES decouples sensed data from their sources. Sensed data is represented by means of spaces. This allows to have different representation of the same data at different abstraction levels using the notion of mapping. The model proposed allows applications to be completely unaware of the physical sensors that acquired the data, and allows different applications to reason on acquired data represented at the correct abstraction level.

5.1.1 Time Driven Sensor Hub

TDSH is a set of architectural abstraction for the design of the components that drives the sensors acquisition. TDSH core concepts include timers, clocks, time-driven activities, and timelines. Timers generate time related events, while clocks keep track of the advances of time as defined by the timers; time-driven activities are activities whose activation is driven by timers, and timelines are data structures that constitute a static representation of time as a numbered sequence of time grains.

A concrete TDSH implementation has been provided, tailored for embedded systems without the support of any real-time Operating System. Moreover, an hardware specific library has been implemented, in order to run TDSH on a *STM32F4-Discovery*, exploiting the HAL driver library to offer features based on the specific hardware, such as ADC converters and the DMA data transmission mode.

Finally, some TDSH practical examples have been produced to verify that the key aspects of acquisition timing and inter-component data management were consistent with the model. Moreover an actual demo application was developed to test TDSH as a separated acquisition component.

5.1.2 Subjective sPaces Architecture for Contextualising hEterogeneous Sources

SPACES is a set of architectural abstractions for standardise sensor measurements representation. The abstractions rely on a *spatial representation* of sensor data. Samples are termed stimuli and localised in terms of measurement spaces and positioning spaces. Mapping functions allow to map stimuli into different spaces enabling other entities to reason on the same data with different representations and at different levels of abstraction.

The concrete implementation of the SPACES approach is tailored for desktop applications and reduces the effort for data fusion and interpretation, while enforcing the reuse of infrastructures. The initial test case for sensor data acquisition has been rethought and expanded to take advantage of the SPACES approach, demonstrating how it enables AAL applications to perform spatio-temporal reasoning.

5.1.3 Publications

The result of this thesis have been published in the following papers:

Journals

- Daniela Micucci, Marco Mobilio, Paolo Napoletano and Francecso Tisato, “Falls as anomalies: an experimental evaluation from smartphone accelerometer data”. Journal of Ambient Intelligence and Humanized Computing: Models and architectures for emergency management (JAIHC) - 2015.

- Daniela Micucci Marco Mobilio and Francesco Tisato, "SPACES: Subjective sPaces Architecture for Contextualizing hEterogeneous Sources". Communications in Computer and Information Science (CCIS) - 2015.

Under revision:

- Micucci, Daniela, Marco Mobilio, and Paolo Napoletano. "UniMiB SHAR: a new dataset for human activity recognition using acceleration data from smartphones". Submitted to IET Electronics Letters - 2016.

Proceedings of International Conferences and Workshops

- Daniela Micucci, Marco Mobilio, Paolo Napoletano and Francecso Tisato, "On the robustness of detecting falls as anomalies from smartphone accelerometer data". Proactive Workshop - 2015.
- Alessio Fanelli, Daniela Micucci, Marco Mobilio, and Francecso Tisato, "Spatio-Temporal Normalization of Data from Heterogeneous Sensors". International Conference on Software Engineering and Applications (ICSOFT-EA) - 2015.
- Marco Mobilio, Toshi Kato, Hiroko Kudo, and Daniela Micucci, "Ambient Assisted Living for an Ageing Society: a Technological Overview". Second Italian Workshop on Artificial Intelligence for Ambient Assisted Living (AI*AAL) - 2016.

The following papers are under developing:

- Daniela Micucci, Marco Mobilio, and Francesco Tisato: "Time Driven Sensor Hub: Time Awareness in Embedded Systems".
- Daniela Micucci, Marco Mobilio, and Paolo Napoletano: "A Survey on Publicly Available ADLs and Falls Datasets".

5.2 Future Developments

For sensor data acquisition, future works may concern improvements in the concrete design of the TDSH, such as the handling of finite-state performers, or an interrupt-based approach to enable performers run to cross the duration of a single tick. Moreover, the synchronisation with upper levels may be further inquired.

Regarding the implementation, further sensors and functionalities could be added to the current library. Moreover hardware specific libraries for different boards may be implemented, to encourage usage.

For data normalisation, the SPACES library of spaces, mapping functions, and membership functions could be expanded to be able to represent more diversified data.

Finally in the field of AAL the fall detection application may be expanded to cover more scenarios and sensors, thus becoming a more complete AAL system.

Bibliography

[1] Johns Hopkins EpiWatch: App and Research Study. `http://www.hopkinsmedicine.org/epiwatch`. 00000.

[2] ISO 9241-11 Guidance on Usability. Technical report, International Organization for Standardization, 1998. 00002.

[3] AAliance2. Ambient Assisted Living Roadmap. Deliverable, AALIANCE2 - European Next Generation Ambient Assisted Living Innovation Alliance, September 2014. 00000.

[4] Emile Aarts and José Encarnação, editors. *True Visions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 00006.

[5] Stefano Abbate, Marco Avvenuti, Paolo Corsini, Janet Light, and Alessio Vecchio. Monitoring of Human Movements for Fall Detection and Activities Recognition in Elderly Care Using Wireless Sensor Network: A Survey. In *Wireless Sensor Networks: Application-Centric Design*. InTech, December 2010. 00000.

[6] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-Awareness. In Hans-W. Gellersen, editor, *Handheld and Ubiquitous Computing*, number 1707 in Lecture Notes in Computer Science, pages 304–307. Springer Berlin Heidelberg, September 1999. 04686.

[7] Gregory D. Abowd and Elizabeth D. Mynatt. Designing for the Human Experience in Smart Environments. In Diane J. Cook and Sajal K. Das, editors, *Smart Environments*, pages 151–174. John Wiley & Sons, Inc., 2004.

[8] Giovanni Acampora, Diane J. Cook, Parisa Rashidi, and Athanasios V. Vasilakos. A Survey on Ambient Intelligence in Healthcare. *Proceedings of the IEEE*, 101(12):2470–2494, December 2013. 00132.

[9] Hamid Aghajan, Juan Carlos Augusto, and Ramon Lopez-Cozar Delgado. *Human-Centric Interfaces for Ambient Intelligence*. Academic Press, September 2009. 00047.

[10] Roberto Alesii, Fabio Graziosi, Stefano Marchesani, Claudia Rinaldi, Marco Santic, and Francesco Tarquini. Advanced Solutions to Support Daily Life of People Affected by the Down Syndrome. In *Ambient Assisted Living*, pages 233–244. Springer, 2015. 00000.

[11] S. Amendola, R. Lodato, S. Manzari, C. Occhiuzzi, and G. Marrocco. RFID Technology for IoT-Based Personal Healthcare in Smart Spaces. *IEEE Internet of Things Journal*, 1(2):144–152, April 2014. 00042.

[12] W. L. Anderson and J. M. Wiener. The Impact of Assistive Technologies on Formal and Informal Home Care. *The Gerontologist*, 55(3):422–433, June 2015. 00006.

[13] Pablo Oliveira Antonino, Daniel Schneider, Cristian Hofmann, and Elisa Yumi Nakagawa. Evaluation of AAL Platforms According to Architecture-Based Quality Attributes. In David V. Keyson, Mary Lou Maher, Norbert Streitz, Adrian Cheok, Juan Carlos Augusto, Reiner Wichert, Gwenn Englebienne, Hamid Aghajan, and Ben J. A. Kröse, editors, *Ambient Intelligence*, Lecture Notes in Computer Science, pages 264–274. Springer Berlin Heidelberg, November 2011. 00018.

[14] Juan Carlos Augusto and Chris D. Nugent. The use of temporal reasoning and management of complex events in smart homes. In *ECAI*, volume 16, page 778. Citeseer, 2004. 00110.

[15] BackHome Partners. BackHome Project. 00000.

[16] R. Baheti and H. Gill. Cyber-physical systems. *Impact Control Technologies*, pages 1–6, 2011. 00233.

[17] Ling Bao and Stephen S. Intille. Activity Recognition from User-Annotated Acceleration Data. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive Computing*, number 3001 in Lecture Notes in Computer Science, pages 1–17. Springer Berlin Heidelberg, April 2004. 02041.

[18] Stephanie Blackman, Claudine Matlo, Charisse Bobrovitskiy, Ashley Waldoch, Mei Lan Fang, Piper Jackson, Alex Mihailidis, Louise Nygård, Arlene Astell, and Andrew Sixsmith. Ambient Assisted Living Technologies for Aging Well: A Scoping Review. *Journal of Intelligent Systems*, 25(1):55–69, 2015. 00002.

[19] Brian M. Bot, Christine Suver, Elias Chaibub Neto, Michael Kellen, Arno Klein, Christopher Bare, Megan Doerr, Abhishek Pratap, John Wilbanks, E. Ray Dorsey, Stephen H. Friend, and Andrew D. Trister. The mPower study, Parkinson disease mobile data collected using ResearchKit. *Scientific Data*, 3, March 2016. 00005.

[20] O. Brdiczka, J. L. Crowley, and P. Reignier. Learning Situation Models in a Smart Home. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(1):56–63, February 2009. 00097.

[21] Davide Calvaresi, Daniel Cesarini, Paolo Sernani, Mauro Marinoni, Aldo Franco Dragoni, and Arnon Sturm. Exploring the ambient assisted living domain: A systematic review. *Journal of Ambient Intelligence and Humanized Computing*, May 2016.

[22] Fabien Cardinaux, Deepayan Bhowmik, Charith Abhayaratne, and Mark S. Hawley. Video based technology for ambient assisted living: A review of the literature. *Journal of Ambient Intelligence and Smart Environments*, 3(3):253–269, 2011. 00067.

[23] Carlos Medrano, Raul Igual, Inmaculada Plaza, and Manuel Castro. Detecting Falls as Novelties in Acceleration Patterns Acquired with Smartphones. *PLoS ONE*, 9(4):e94811, April 2014. 00010.

[24] Amedeo Cesta, Gabriella Cortellessa, Riccardo Rasconi, Federico Pecora, Massimiliano Scopelliti, and Lorenza Tiberio. Monitoring Elderly People with the Robocare Domestic Environment: Interaction Synthesis and User Evaluation. *Computational Intelligence*, 27(1):60–82, February 2011. 00044.

[25] Jay Chen, Karric Kwong, Dennis Chang, Jerry Luk, and Ruzena Bajcsy. Wearable sensors for reliable fall detection. In *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 3551–3554. IEEE, 2006. 00272.

[26] Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Building Virtual Sensors and Actuators over Logical Neighborhoods. In *Proceedings of the International Workshop on Middleware for Sensor Networks*, MidSens '06, pages 19–24, New York, NY, USA, 2006. ACM. 00033.

[27] Diane J. Cook, Juan C. Augusto, and Vikramaditya R. Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298, August 2009. 00540.

[28] Diane J. Cook, Aaron S. Crandall, Brian L. Thomas, and Narayanan C. Krishnan. CASAS: A smart home in a box. *Computer*, 46(7), 2013. 00065.

[29] Cecile KM Crutzen. Invisibility and the meaning of ambient intelligence. *International Review of Information Ethics*, 6(12):52–62, 2006. 00025.

[30] Jiangpeng Dai, Xiaole Bai, Zhimin Yang, Zhaohui Shen, and Dong Xuan. PerFallD: A pervasive fall detection system using mobile phones. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 292–297. IEEE, 2010. 00117.

[31] Arnaldo D'Amico and Corrado Di Natale. Beyond Human Senses: Technologies, Strategies, Opportunities, and New Responsibilities. In *Sensors*, pages 3–7. Springer, 2014. 00000.

[32] Ranjan Dasgupta and Shuvashis Dey. A comprehensive sensor taxonomy and semantic knowledge representation: Energy meter use case. In *Sensing Technology (ICST), 2013 Seventh International Conference on*, pages 791–799. IEEE, 2013. 00009.

[33] Fred D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13(3):319–340, 1989. 28712.

[34] Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. *Scenarios for Ambient Intelligence in 2010*. Office for official publications of the European Communities, 2001. 00681.

[35] October Duke University. Duke Launches Autism Research App. `https://today.duke.edu/2015/10/autismbeyond`, October 2015. 00000.

[36] Nicholas Farber, Public Policy Institute (AARP (Organization)), and National Conference of State Legislatures. *Aging in Place: A State Survey of Livability Policies and Practices*. AARP Public Policy Institute ; National Conference of State Legislatures, Washington, D.C.; Denver, Colo., 2011. 00049.

[37] Francesco Fiamberti, Daniela Micucci, Alessandro Morniroli, and Francesco Tisato. A Model for Time-Awareness. In *SpringerLink*, pages 70–84. Springer Berlin Heidelberg. 00001.

[38] A. Fleury, M. Vacher, and N. Noury. SVM-Based Multimodal Classification of Activities of Daily Living in Health Smart Homes: Sensors, Algorithms, and First Experimental Results. *IEEE Transactions on Information Technology in Biomedicine*, 14(2):274–283, March 2010. 00202.

[39] Giancarlo Fortino, Anna Rovella, Wilma Russo, and Claudio Savaglio. On the Classification of Cyberphysical Smart Objects in the Internet of Things. In *UBICITEC*, pages 86–94, 2014. 00005.

[40] Dov M Gabbay and John Alan Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming: Volume 5: Logic Programming*. Clarendon Press, 1998. 00453.

[41] Nuno M. Garcia and Joel Jose P. C. Rodrigues. *Ambient Assisted Living*. CRC Press, June 2015. 00028.

[42] Jonathan Grant, Stijn Hoorens, Suja Sivadasan, Mirjam van het Loo, Julie DaVanzo, Lauren Hale, Shawna Gibson, and William Butz. Low Fertility and Population Ageing. `http://www.rand.org/pubs/monographs/MG206.html`, 2004. 00131.

[43] Trisha Greenhalgh, Sara Shaw, Joe Wherton, Gemma Hughes, Jenni Lynch, Christine A'Court, Sue Hinder, Nick Fahy, Emma Byrne, Alexander Finlayson, Tom Sorell, Rob Procter, and Rob Stones. SCALS: A fourth-generation study of assisted living technologies in their organisational, social, political and policy context. *BMJ Open*, 6(2):e010208, January 2016. 00000.

[44] Levent Gurgen, Claudia Roncancio, Cyril Labbé, André Bottaro, and Vincent Olive. SStreaMWare: A service oriented middleware for heterogeneous sensor data management. In *Proceedings of the 5th International Conference on Pervasive Services*, pages 121–130. ACM, 2008. 00083.

[45] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, number 2211 in Lecture Notes in Computer Science, pages 166–184. Springer Berlin Heidelberg, October 2001. 00374.

[46] Brandon Ballinger Hsieh, Johnson. Can deep neural networks save your neural network? artificial intelligence, sensors, and strokes: Big data conference: Strata + Hadoop World, March 28 - 31, 2016, San Jose, CA. `http://conferences.oreilly.com/strata/hadoop-big-data-ca/public/schedule/detail/47144`. 00000.

[47] V. Jeet, H. S. Dhillon, and S. Bhatia. Radio Frequency Home Appliance Control Based on Head Tracking and Voice Control for Disabled Person. In *2015 Fifth International Conference on Communication Systems and Network Technologies (CSNT)*, pages 559–563, April 2015. 00000.

[48] Jie Yin, Qiang Yang, and J.J. Pan. Sensor-Based Abnormal Human-Activity Detection. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):1082–1090, August 2008. 00146.

[49] Emil Jovanov, Aleksandar Milenkovic, Chris Otto, and Piet C De Groen. A wireless body area network of intelligent motion sensors for computer assisted physical rehabilitation. *Journal of NeuroEngineering and rehabilitation*, 2(1):6, 2005. 00847.

[50] Mayank Kaushik, Matthew Trinkle, Ahmad Hashemi-Sakhtsari, and Tim Pattison. Three dimensional microphone and source position estimation

using TDOA and TOF measurements. In *Signal Processing, Communications and Computing (ICSPCC), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011. 00001.

[51] Kensaku Kawamoto, Caitlin A. Houlihan, E. Andrew Balas, and David F. Lobach. Improving clinical practice using clinical decision support systems: A systematic review of trials to identify features critical to success. *BMJ*, 330(7494):765, March 2005. 01555.

[52] Thomas Kleinberger, Martin Becker, Eric Ras, Andreas Holzinger, and Paul Müller. Ambient intelligence in assisted living: Enable elderly people to handle future interfaces. In *Universal Access in Human-Computer Interaction. Ambient Interaction*, pages 103–112. Springer, 2007. 00232.

[53] J. Klenk, C. Becker, F. Lieken, S. Nicolai, W. Maetzler, W. Alt, W. Zijlstra, J. M. Hausdorff, R. C. van Lummel, L. Chiari, and U. Lindemann. Comparison of acceleration signals of simulated and real-world backward falls. *Medical Engineering & Physics*, 33(3):368–373, April 2011. 00066.

[54] E. A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008. 01523.

[55] Jay Lee, Behrad Bagheri, and Hung-An Kao. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23, January 2015. 00111.

[56] Qiang Li, John A. Stankovic, Mark A. Hanson, Adam T. Barth, John Lach, and Gang Zhou. Accurate, fast fall detection using gyroscopes and accelerometer-derived posture information. In *Wearable and Implantable Body Sensor Networks, 2009. BSN 2009. Sixth International Workshop on*, pages 138–143. IEEE, 2009. 00297.

[57] Yun Li, K.C. Ho, and M. Popescu. A Microphone Array System for Automatic Fall Detection. *IEEE Transactions on Biomedical Engineering*, 59(5):1291–1301, May 2012. 00083.

[58] Liming Chen, J. Hoey, C. D. Nugent, D. J. Cook, and Zhiwen Yu. Sensor-Based Activity Recognition. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):790–808, November 2012. 00158.

[59] Leili Lind, Gunnar Carlgren, and Daniel Karlsson. Old—and With Severe Heart Failure: Telemonitoring by Using Digital Pen Technology in Specialized Homecare. *CIN: Computers, Informatics, Nursing*, page 1, May 2016. 00000.

[60] Yong Liu, David Hill, Alejandro Rodriguez, Luigi Marini, Rob Kooper, James Myers, Xiaowen Wu, and Barbara Minsker. A new framework

for on-demand virtualization, repurposing and fusion of heterogeneous sensors. pages 54–63. IEEE, 2009. 00018.

[61] Inês P. Machado, A. Luísa Gomes, Hugo Gamboa, Vítor Paixão, and Rui M. Costa. Human activity data discovery from triaxial accelerometer sensor: Non-supervised learning sensitivity to feature extraction parametrization. *Information Processing & Management*, 51(2):204–214, March 2015. 00006.

[62] U. Maurer, A. Smailagic, D. P. Siewiorek, and M. Deisher. Activity recognition and monitoring using multiple sensors on different body positions. In *International Workshop on Wearable and Implantable Body Sensor Networks (BSN'06)*, pages 4 pp.–116, April 2006. 00419.

[63] Daniela Micucci, Marco Mobilio, Paolo Napoletano, and Francesco Tisato. Falls as anomalies? An experimental evaluation using smartphone accelerometer data. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–13, December 2015. 00000.

[64] Daniela Micucci, Marco Mobilio, and Francesco Tisato. SPACES: Subjective sPaces Architecture for Contextualizing hEterogeneous Sources. In Pascal Lorenz, Jorge Cardoso, Leszek A. Maciaszek, and Marten van Sinderen, editors, *Software Technologies*, number 586 in Communications in Computer and Information Science, pages 415–429. Springer International Publishing, July 2015. 00000.

[65] Felip Miralles, Eloisa Vargiu, Stefan Dauwalder, Marc Sola, Juan Manuel Fernández, Eloi Casals, and José Alejandro Cordero. Telemonitoring and home support in backhome. *Information Filtering and Retrieval*, page 24, 2014. 00003.

[66] S. M. R. Moosavi and A. Sadeghi-Niaraki. A SURVEY OF SMART ELECTRICAL BOARDS IN UBIQUITOUS SENSOR NETWORKS FOR GEOMATICS APPLICATIONS. In *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume XL-1-W5, pages 503–507. Copernicus GmbH, December 2015. 00000.

[67] Giovanna Morgavi, Roberto Nerino, Lucia Marconi, Paola Cutugno, Claudia Ferraris, Alessandra Cinini, and Mauro Morando. An Integrated Approach to the Well-Being of the Elderly People at Home. In *Ambient Assisted Living*, pages 265–274. Springer, 2015. 00001.

[68] Muhammad Mubashir, Ling Shao, and Luke Seed. A survey on fall detection: Principles and approaches. *Neurocomputing*, 100:144–152, January 2013. 00202.

[69] Hans Inge Myrhaug. Towards life-long and personal context spaces. In *Workshop on User Modelling for Context-Aware Applications*, 2001. 00009.

[70] S. Nefti, U. Manzoor, and S. Manzoor. Cognitive agent based intelligent warning system to monitor patients suffering from dementia using ambient assisted living. In *2010 International Conference on Information Society (I-Society)*, pages 92–97, June 2010. 00018.

[71] Gbenga Ogedegbe and Thomas Pickering. Principles and Techniques of Blood Pressure Measurement. *Cardiology Clinics*, 28(4):571–586, November 2010. 00049.

[72] Smitha Paulose, E Sebastian, and B Paul. Acoustic source localization. *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, 2(2):933–9, 2013. 00005.

[73] Maria-Salome Perez and Enrique V. Carrera. Acoustic event localization on an Arduino-based wireless sensor network. In *Communications (LATINCOM), 2014 IEEE Latin-America Conference on*, pages 1–6. IEEE, 2014. 00002.

[74] Philips Electronics. Building Technology for People. `http://www.newscenter.philips.com/pwc_nc/main/shared/assets/Downloadablefile/CEBIT-Asia-Keynote-speech-Van-Splunter(1)-3734-1440.pdf`, February 2002. 00000.

[75] Gerald Pirkl, Daniele Munaretto, Carl Fischer, Chunlei An, Paul Lukowicz, Martin Klepal, Andreas Timm-Giel, Joerg Widmer, Dirk Pesch, Hans Gellersen, and others. Virtual lifeline: Multimodal sensor data fusion for robust navigation in unknown environments. *Pervasive and Mobile Computing*, 8(3):388–401, 2012. 00012.

[76] M. Popescu, Yun Li, M. Skubic, and M. Rantz. An acoustic fall detector system that uses sound height information to reduce the false alarm rate. In *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2008. EMBS 2008*, pages 4628–4631, August 2008. 00099.

[77] M. Popescu and A. Mahnot. Acoustic fall detection using one-class classifiers. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2009. EMBC 2009*, pages 3505–3508, September 2009. 00023.

[78] Proteus Digital Health. Proteus Digital Health Announces FDA Clearance of Ingestible Sensor. `http://proteusdigitalhealth.com/proteus-digital-health-announces-fda-clearance-of-%0020ingestible-sensor/`, 2012. 00000.

[79] Jörg Rech and Klaus-Dieter Althoff. Artificial Intelligence and Software Engineering: Status and Future Trends. *Special Issue on Artificial Intelligence and Software Engineering, KI*, 3:5–11, 2004. 00032.

[80] Dori Rosenberg, Colin A. Depp, Ipsit V. Vahia, Jennifer Reichstadt, Barton W. Palmer, Jacqueline Kerr, Greg Norman, and Dilip V. Jeste. Exergames for Subsyndromal Depression in Older Adults: A Pilot Study of a Novel Intervention. *The American Journal of Geriatric Psychiatry*, 18(3):221–226, March 2010. 00184.

[81] C. Rougier, J. Meunier, A. St-Arnaud, and J. Rousseau. Robust Video Surveillance for Fall Detection Based on Human Shape Deformation. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(5):611–622, May 2011. 00147.

[82] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *First Workshop on Mobile Computing Systems and Applications, 1994. WMCSA 1994*, pages 85–90, December 1994. 04031.

[83] Albrecht Schmidt. Interactive context-aware systems interacting with ambient intelligence. *Ambient intelligence*, 159, 2005. 00154.

[84] AJ Sixsmith. An evaluation of an intelligent home monitoring system. *Journal of telemedicine and telecare*, 6(2):63–72, 2000. 00156.

[85] Frank Sposaro and Gary Tyson. iFall: An Android application for fall monitoring and response. In *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pages 6119–6122. IEEE, 2009. 00195.

[86] Alan N. Steinberg, Christopher L. Bowman, and Franklin E. White. Revisions to the JDL data fusion model. volume 3719, pages 430–441, 1999. 00802.

[87] Robert J. Stone. Haptic feedback: A brief history from telepresence to virtual reality. In Stephen Brewster and Roderick Murray-Smith, editors, *Haptic Human-Computer Interaction*, number 2058 in Lecture Notes in Computer Science, pages 1–16. Springer Berlin Heidelberg, 2001. 00161.

[88] Ying Tan, Steve Goddard, and Lance C. Pérez. A Prototype Architecture for Cyber-physical Systems. *SIGBED Rev.*, 5(1):26:1–26:2, January 2008. 00113.

[89] Emmanuel Munguia Tapia, Stephen S. Intille, and Kent Larson. Activity Recognition in the Home Using Simple and Ubiquitous Sensors. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive Computing*, number 3001 in Lecture Notes in Computer Science, pages 158–175. Springer Berlin Heidelberg, April 2004. 01058.

[90] Mary E. Tinetti, Mark Speechley, and Sandra F. Ginter. Risk Factors for Falls among Elderly Persons Living in the Community. *New England Journal of Medicine*, 319(26):1701–1707, December 1988. 05378.

[91] Francesco Tisato, Carla Simone, Diego Bernini, Marco P. Locatelli, and Daniela Micucci. Grounding ecologies on multiple spaces. *Pervasive and Mobile Computing*, 8(4):575–596, August 2012. 00010.

[92] UN. World Population Ageing 2013. Technical report, 2013. 00002.

[93] Rob van Ommering. Building Product Populations with Software Components. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 255–265, New York, NY, USA, 2002. ACM. 00223.

[94] Athanasios Vasilakos and Witold Pedrycz. *Ambient Intelligence, Wireless Networking, And Ubiquitous Computing*. Artech House, Inc., Norwood, MA, USA, 2006. 00074.

[95] Viswanath Venkatesh, Michael G. Morris, Gordon B. Davis, and Fred D. Davis. User Acceptance of Information Technology: Toward a Unified View. *MIS Quarterly*, 27(3):425–478, 2003. 13748.

[96] M. Weiser. Hot topics-ubiquitous computing. *Computer*, 26(10):71–72, October 1993. 00749.

[97] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991. 13446.

[98] D. H. Wilson and C. Atkeson. Simultaneous Tracking and Activity Recognition (STAR) Using Many Anonymous, Binary Sensors. In Hans-W. Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive Computing*, number 3468 in Lecture Notes in Computer Science, pages 62–79. Springer Berlin Heidelberg, May 2005. 00257.

[99] Niklaus Wirth. Toward a Discipline of Real-time Programming. *Commun. ACM*, 20(8):577–583, August 1977. 00201.

[100] Carlos A. Zarate, Jr, Lisa Weinstock, Peter Cukor, Cassandra Morabito, Linda Leahy, and Lee Baer. Applicability of Telemedicine for Assessing Patients With Schizophrenia: Acceptance and Reliability. *The Journal of Clinical Psychiatry*, 58(1):22–25, January 1997. 00167.