Ref. Ares(2015)6068541 - 31/12/2015

Health, demographic change and wellbeing

Personalising health and care: Advancing active and healthy ageing

H2020-PHC-19-2014

Research and Innovation Action

ACANTO

A CyberphysicAl social NeTwOrk using robot friends

Deliverable 7.1.1

Deliverable Title

Deliverable due date: 11.2015	**Actual submission date: day.month.year**
Start date of project: February 1, 2015	**Duration: 42 months**
Lead beneficiary for this deliverable: TI	**Revision: 1.2**
Authors:Gian Piero Fici (TI), Luigi Palopoli (UNITN), Carlos Rivera (ENVT), Ivo Ramos (ATOS)	
Internal reviewer: NA	

The research leading to these results has received funding from the European Union's H2020 Research and Innovation Programme - Societal Challenge 1 (DG CONNECT/H) under grant agreement n°643644

Dissemination Level

PU	Public	
CO	Confidential, only for members of the consortium (including the Commission Services)	**X**

The contents of this deliverable reflect only the authors' views and the European Union is not liable for any use that may be made of the information contained therein.

Contents

Executive Summary

This document contains the definition of APIs defined between the different software components of the ACANTO architecture.

Since the accurate definition of the APIs follows the implementation of the overall system the present version of this deliverable is not to be considered the final description of the data structures and of the interactions between modules composing the ACANTO architecture. On the contrary, the document contains our initial design choices on the API, which are stated in this early phase of the project in order to facilitate the subsequent integration steps. For this reason, the ideas expressed in the document are subject to constant revision and adaptation as required by the implementation and integration activities.

In its central part, the document describes the software interfaces for the execution of social activities. Since low fidelity prototypes have already been developed at this stage to the purpose of facilitating requirement collection, we regard the design choices expressed therein as quite stable.

The document utilizes the UML modeling language whenever possible and assumes a C++ binding. An overarching choice of our implementation is the use of template meta-programming as a means to obtain correct by construction code. Therefore, we shall frequently refer to template classes, for which standard UML support is limited. The document will provide high-level description for the methods, with the primary goal of describing the semantics, and the interactions between the classes rather than their exact syntax. For implementation purposes, this document will go hand-in-hand with the Doxygen documentation of the classes and of the functions, which can be found in the project git repositories and is subject to continuous revision.

In the first part of the document the ACANTO architecture is described. Then the interfaces between all the components are defined. In particular the document is divided following the main subsystems of the ACANTO architecture, i.e.: Conception of social activities, Execution of social activities, Perception of user and environment. For each of these subsystems the data structures used to store and exchange information and the class diagrams of the implementation are described following the needs of the specific module addressed.

Chapter 1 - The ACANTO architecture

The following figure shows the main functional blocks of ACANTO and illustrates their interconnection.

A number of subsystems are identified:

- Conception of social activities
- Execution of social activities
- Perception of user and environment

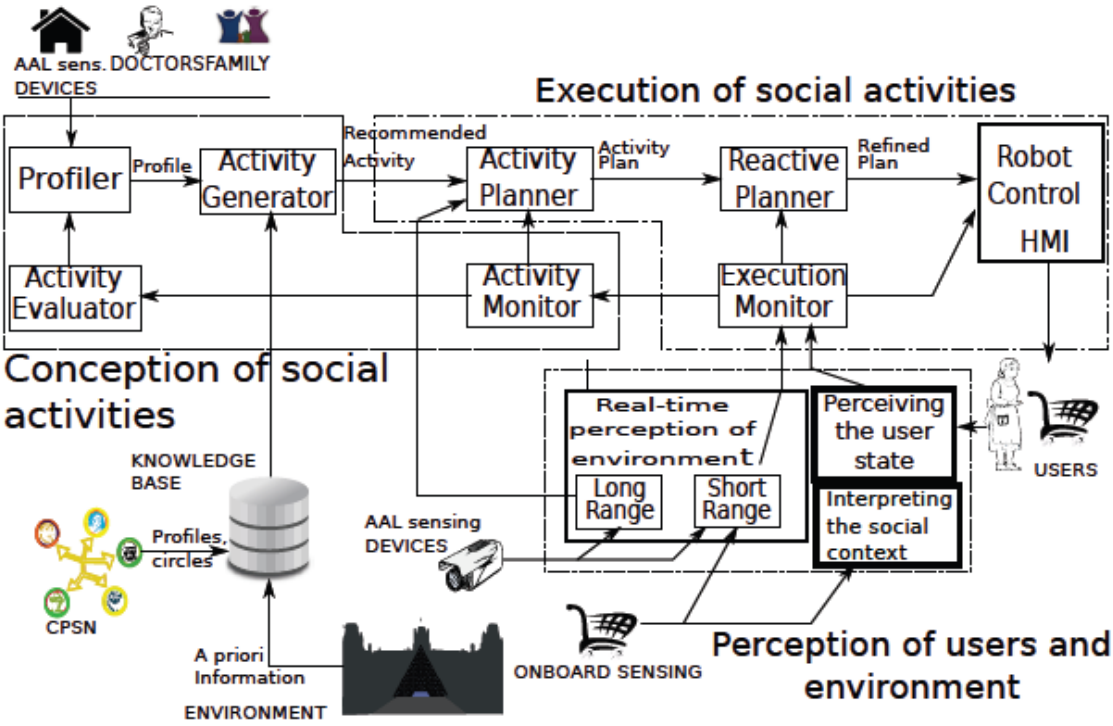

Figure 1: The ACANTO overall architecture

The first subsystem that we identify in the figure is responsible for the **Conception of social activities**. Within this subsystem, the `Profiler`, collects relevant information on the user from different sources and consolidates it with the past observations to create and update the profile. The `Activity Generator` decides recommendations on possible activities. This is done accounting for the preferences of other people belonging to the circles of the user, for the state of the user (e.g., the level of supplies in the pantry) and for the opportunities offered by the environment (e.g., special events planned for the day, discounts offered by a mall etc.). The `Activity Monitor` monitors the progress of an activity (e.g., the number of planned locations that have been visited) and decides changes to the activity in case of significant deviations (e.g., the user is too tired to proceed) or of unpredicted events (e.g., one of the locations is not available or is overcrowded). The `Activity Evaluator` comes into play at the end of an activity. It assesses the therapeutic efficacy of the activity, the user's satisfaction, as judged from her/his physical and emotional state observed during the different phases, and the quality of social interactions within the circle (if the activity involves more than one user). This information is used to refine the profile and evaluate the possibility of new circles.

The second subsystem is in charge of the **Execution of social activities**. One of its components, the `Activity Planner`, decides a plan to execute the activity accounting for the state of the environment observed by remote sensors and by devices carried by other users connected to the system. The plan specifies a schedule with places to visit, tasks to perform and

comfort properties that need to be preserved all the while. The `Reactive Planner` exploits information collected by the sensing system and integrated by the execution monitor, to refine the plan and turn it into an executable sequence of actions. In particular, it produces a motion plan that: 1. secures a correct execution of the physical exercise planned for each user, 2. respects the social rules, 3. guarantees sufficient safety levels and keeps the user's stress in check. When the activity is carried out in group, the plan has to guarantee a proper coordination of the different elements and account for the social dynamics and roles within the group. The execution of the plan is supported by a multi-modal Human Machine Interface (HMI), which makes use of displays, haptic devices (wearable or embedded in the grips of the FriWalk) and robotic mechanical guidance. These devices operate in closed loop, using the information collected in real–time from the user and from the environment and integrated by the execution monitor. When anomalies that could impair the execution of the plan are detected (e.g., a group of bystanders on the path, or a wet floor sign), the `Reactive Planner` explores possible minimal deviations to the plan that will nevertheless respect the high level objectives (set by the `Activity Planner`), and preserve the required properties. If this is possible, the adjustment is implemented; otherwise, an exception is signaled to the `Activity Planner`, which tries to produce a new high level plan (e.g., re-shuffling the execution of the activities). If the changes in the environment or in state of user the show no possibility of solution, the exception has to be propagated to the `Activity Generator`, which changes the planned activity.

The third subsystem is in charge of **Perception of users and environment**. It combines algorithms for sensing and for extracting semantic information from the sensed data. The sources of information are manifold and are partially implemented onboard the FriWalk and worn by the user, and partially in the environment. Onboard and wearable sensors are used to observe the user and to gauge her/his physical and emotional state. They are also used to detect the environment in the proximity of the user (short range sensors). Sensors in the environment allow the system to extend its perception range. Possible applications are to identify overcrowded areas and to pinpoint the position of security staff and assistants in advance. The planned path can be chosen in order to minimize the stress of sharing busy spaces and to enhance the user's sense of security by remaining within sight of personnel able to offer prompt assistance if required.

Chapter 2- Conception of social activities

In this chapter the two main data structure of the **Conception of social activities** subsystem are described: Profile and Activity.
These data structures are created within this subsystem and are used for the internal communication between modules within this same subsystem and for the external communication between this subsystem and the **Execution of social activities** subsystem.

2.1 Profile

The Profile data structure describes the profile of the user of the ACANTO system. It is composed of several subclasses. Three dots mean that the size is not fixed and can be extended. Sets are similar to Enums but they can contain more than one identifier simultaneously. Also these identifiers may be classes themselves. The following is not written using a strictly formal language but it is more of a pseudo-code definition:

Class name	Class Description
Person	String name; Date date_of_birth; String e_mail;
Characteristic	Int height; Int weight; Enum { married, single, divorced, widow } marital_status;
Ability	Set < Goodhealth, Balance_problems, Mobility_problems, ...> Where.... Set Mobility_problems <Arthritis, ...>;
Interest	Set <Sports, Cooking, Gardening, ...>
Activity	Activity* activities; (array of Activity, see below) Time* time_on_activity; (array of Time)

Education	Set < None, High_school, Community_college, Undergraduate_degree, PHd > degree; Set < Spanish, English, Italian, Greek, German, Esperanto, ... > languages;
Profession	Set < Doctor, Policeman, Firefighter, Bureaucrat, Bicycle_messenger, ...>;
Living Conditions	Set < Public_transport_available, Private_transport_available, Stairs_at_home, ...>;
Contact Other persons	Person* (array of Person class)
Preference	Set < Cats, Classical_music, Blue_color, ...> likes; Set < Cats, Classical_music, Blue_color, ...> dislikes;
Social networks information	Set < Struct <name, url>, ... >;
Constrains	Set <Barriers, ...> barriers_to_mobility; Set <Walking_frame, Walking_sticks, Stair_lift, ...> mobility_aids;
Health prescriptions	Set <Walk, SitUps, ...>; Where for example Walk is defined as: Struct Walk { Int distance, Time min_time, ... };

2.2 Activity

The Activity data structure describes the actions the user of the ACANTO system has to perform and the targets he or she has to achieve.

There are three types of activities:

- Rehabilitation activities
  Two sub-categories:
  - Controlled environment (e.g., Hospital, Rehabilitation Center): specific set of exercises supervised by the medical professionals

  - Outside the Hospital: walk x meters per day, which can also be made at the same time that the user visits a museum or a shopping mall
- Equipped indoor environments
  Examples:
  - go to the museum
  - shopping tour
- Outdoor activities
  Examples:
  - meet friends
  - walk in the park

The activities are created by the module `Activity Generator`, and their generation is accomplished accounting for the preferences of other people belonging to the circles of the user, for the state of the user (e.g., the level of supplies in the pantry) and for the opportunities offered by the environment (e.g., special events planned for the day, discounts offered by a mall etc.)

While the attributes of the Activity are initially provided by the user and integrated by care-givers and relatives, they are continuously and automatically updated using the harvested information from the `Activity Evaluator`, which comes into play at the end of an activity. It assesses the therapeutic efficacy of the activity, the user's satisfaction, as judged from her/his physical and emotional state observed during the different phases, and the quality of social interactions within the circle (if the activity involves more than one user). This information is used to refine the profile and evaluate the possibility of new circles.

The Acvitiy data structure is composed of several subclasses. Three dots mean that the size is not fixed and can be extended. Sets are similar to Enums but they can contain more than one identifier simultaneously. Also these identifiers may be classes themselves. The following is not written using a strictly formal language but it is more of a pseudo-code definition:

Class name	**Class Description**
Person	String name; Date date_of_birth; String e_mail;
Characteristic	Int height; Int weight; Enum { married, single, divorced, widow } marital_status;
Interest	Set <Sports, Cooking, Gardening, ...>
Interests of people belonging to the circles of the user	Set <Sports, Cooking, Gardening, ...>

Education	Set < None, High_school, Community_college, Undergraduate_degree, PHd > degree; Set < Spanish, English, Italian, Greek, German, Esperanto, ... > languages;
Profession	Set < Doctor, Policeman, Firefighter, Bureaucrat, Bicycle_messenger, ...>;
Living Conditions	Set < Public_transport_available, Private_transport_available, Stairs_at_home, ...>;
Contact Other persons	Person* (array of Person class)
Preference	Set < Cats, Classical_music, Blue_color, ...> likes; Set < Cats, Classical_music, Blue_color, ...> dislikes;
Social networks information	Set < Struct <name, url>, ... >;
Constrains	Set <Barriers, ...> barriers_to_mobility; Set <Walking_frame, Walking_sticks, Stair_lift, ...> mobility_aids;
Health prescriptions	Set <Walk, SitUps, ...>; Where for example Walk is defined as: Struct Walk { Int distance, Time min_time, ... };

Chapter 3 - Execution of social activities

In this chapter the modules and the internal interfaces of the Execution of social activities subsystem are described.

3.1 Activity Planner

The activity planner is a module that generates a sequence of paths to implement an activity.

In general, the Activity Planner decides the path to follow (given a set of high level constraints and goals). For the specific rehab application, we could also have a direct generation of a path from a caregiver using a specific tool (e.g., an editor).

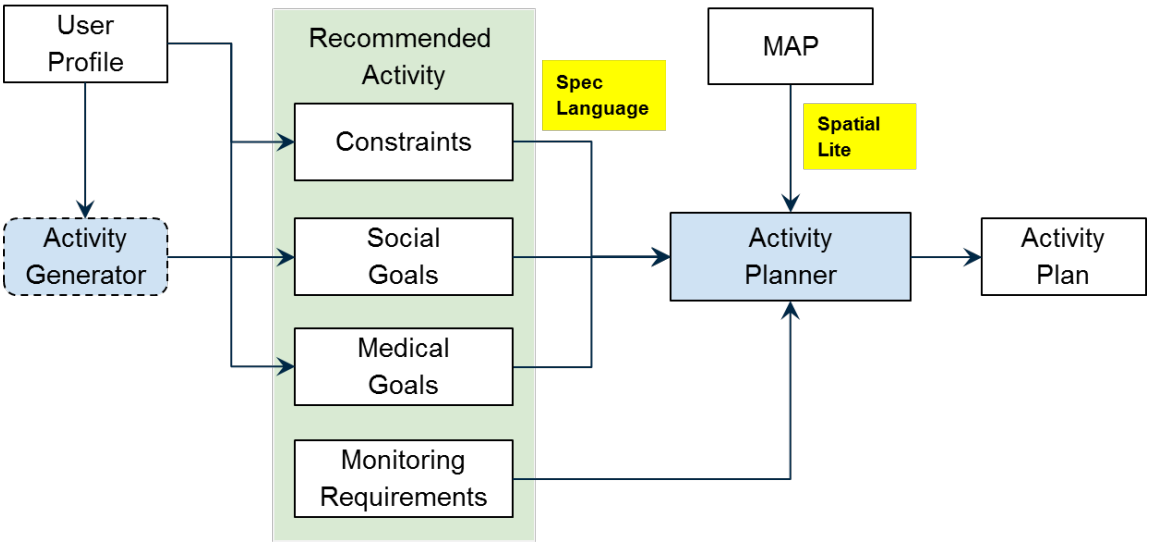

Figure 1: The Activity Planner components

The meaning of the different blocks is as follows:

- **User Profile**: contains all the information concerning a user. It is compounded of a "medical" part (which is strictly confidential) and a "social" part, which derives from previous observations and from the participation to the social network
- **Activity Generator**: is a module that produces a recommendation on an activity to do. For the clinical scenarios the decision on the activity can be directly made by the caregiver.
- **Constraints**: encapsulate all the preferences of the user and other information deriving from the specific scenario (e.g., opening hours of an exhibition)
- **Social Goals**: this is a provisional name to denote the particular goals of the activity (e.g., buying shoes, or visiting renaissance exhibition)
- **Medical Goals**: specify the rehab or prevention objectives related to an activity (e.g., the user has to walk for half an hour at 1.2 m/s average speed).
- **Monitoring Requirements**: contains all the things needed to be observed and reported on the execution of the activity (e.g., duration, fulfilment of the goals, emotional state throughout the activity, etc.).
- **Activity Planner**: generates an executable plan for the activity (as specified below). All the input information that the planner receives are specified in a

formal language to be defined. In our first iteration we will adapt DALi's LT planner for this.

The specification language to express an activity, along with user profile, constraints, goals and requirements is covered in the deliverables produced by WP2 (i.e., D2.3 for the preliminary version, and D2.4 for the final one).

3.2 Activity Execution

The specific representation of a plan is going to be finally specified in WP2 (D2.5 and D2.6). In the present document we offer a very high level view on its required content, as it resulted from the discussions internal to the consortium during the first year of activity of the project.

As documented in the scheme below, an Activity Plan is a sequence of tasks. Each task is associated with a Path, a set of guidance algorithms that have to be used for its execution and the specification of the user's preferences (which could be different for each task).

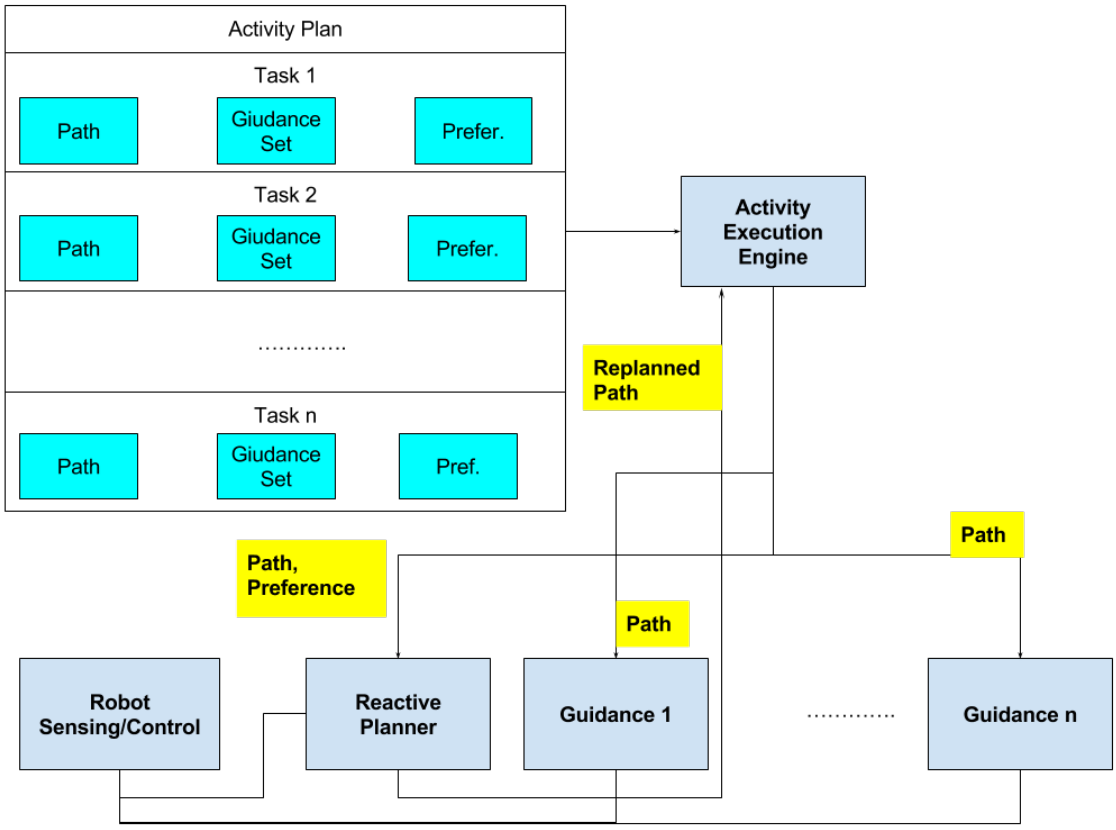

Figure 2: The activity Execution Components

A path is a sequence of waypoints, each one characterised by a couple of coordinates and by a reference time (which is the planned time to reach the waypoint).

The execution of an activity plan is orchestrated by the Activity Execution Engine, which reads the tasks in sequence and hands over the execution of each task to all the guidance algorithms active for the task. The Path is also given to the Reactive Planner, along with the user's preferences. As soon as a violation of the preferences is detected, the planner computes a new path and notifies the Engine.

3.3 Activity Execution: API specification

In the following sections we report and explain the class diagrams of the main components used for the activity execution.

3.3.1 Activity Plan

The first component that we describe in some detail is the *ActivityPlan*, which is used for the execution of any new activity.

The Class diagram of the activity plan is depicted in Figure 4.

The main components that we identify are the following:

1. *AcantoTime* is a template class, with the time granularity as parameter (which is an enumerated type). The use of a template of this type allows us to identify unambiguously the unit that we use for time measurements and avoid possible errors during the compilation phase. Possible values for the granularity are *milliseconds, microseconds, nanoseconds.*
2. *Path:* a path is a sequence of *WayPoint.* Each *WayPoint* is associated with its *x* and *y* coordinates, its curvature and its reference time (i.e., the time where the point is supposedly reached). The *Path* is a parametric class, with the *Smoother* being its template parameter. The idea is that a *Path* is specified through a relatively small number of points. The smoother is then used to generate a smooth Path that is executable (by a *GuidanceAlgorithm*). The operations associated with the *Path* are:
   - *addPoint:* inserts a point in the path used by the *Path* creator to insert a new way point or by the *ReactivePlanner* to insert a new point for a detour.
   - *removePoint:* used by the *ReactivePlanner* to change the shape of a path in response to a perceived violation of the constraints encoded in the *UserPreference*.
   - *getClosestPoint:* called by the guidance algorithm to find the closest element of the path that it is following.
3. *UserPreference:* is a class encoding the preferences related to the specific *Task* underway. Such preferences are related to the individual preferences (e.g., physical limitations, requirements to avoid overcrowded areas) and to the group preferences (e.g., preferred motion patterns for the group). All of these are the outcomes of WP2. It is worth noting preferences are already considered when planning an Activity (e.g., to decide which places to visit). In this case, they are accounted for by the *ReactivePlanner* to decide if the *Task's* goal can be met or if a change is required.
4. *GuidanceIndex*: is an enumerated class that lists possible values for the *GuidanceAlgorithm.*
5. *Task* is a class composed of a *Path,* an instance of *UserPreference* and of a set of *GuidanceIndex*. The rationale is that in each task one ore more guidance solutions can be applied. For instance it is possible to apply *MechanicalFrontWheel, Visual,* and *Haptic* at the same time.

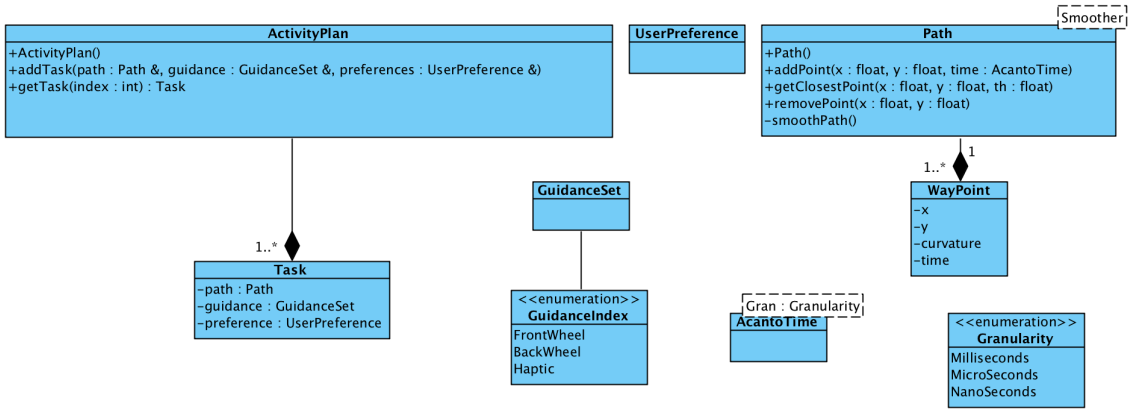

Figure 3: Class Diagram of the activity plan

3.3.2 *ActivityExecutionEngine*: high level view

At the heart of any execution are three classes: the *ActivityExecutionEngine*, the *ReactivePlanner* and the *GuidanceAlgorithm*. The class diagram in Figure 5 describes these classes and their relations.

The *ActivityExecutionEngine* has the following methods:

- *setActivityPlan*: used to initiate a new activity linking the *ActivityExecutionEngine* with a plan that needs to be executed.
- *start*: used to start the execution of an activity.
- *stop*: used to halt the execution of an activity.
- *getActivityState*: it takes a snapshot of the current state in the execution of an activity
- *taskCompleted*: it is used to notify to an application that a task has been completed.
- *taskAborted*: it is used to notify that the execution of a plan is no longer possible within the constraints encoded within the *UserPreference*.

The *ActivityObserver* is an abstract class, which is the root of a hierarchy to define different observers. An observer is used to keep track of all relevant events related to an activity. Observers have to provide a concrete implementation for the method:

- *notify:* used by the observed entity (in this case the *ActivityExecutionEngine*) whenever a relevant event occurs. In response to a *notify* call the observer samples the state of the observed entity and notes the event (e.g., it is possible to define an observer to measure the average duration of a task, the changes of mood of the assisted person, etc.).

GuidanceAlgorithm: is the root of a hierarchy of classes implementing the guidance algorithms. *GuidanceAlgorithm* is an active class (meaning that a thread is associated with it that executes all its mandated operations). Possible specialisation are *MechanicalFrontWheels, MechanicalBackWheels, HapticGuidance.* The class is a template of the *Period* and of its internal state. Every different specialisation requires that the internal state be specialised to a specific value (dependent on the type of state the algorithm manipulates). Each guidance solution in one-to-one correspondence with a guidance index. The algorithm exploits one or more instances of the *Sensor* class to receive the data from the plant and one ore more instances of the *Actuator* class to send the data to the actuator. The idea is that once every *Period* the *GuidanceAlgorithm* is executed: it sample the sensors, decides the control action and generates and action request for the actuators required by the specific algorithm. A *GuidanceAlgorithm* can optionally be linked to an observer (more precisely a PathObserver), which is invoked through the *notify* method whenever a new event occurs (e.g., the awakening of the thread executing the guidance algorithm). The *PathObserver* is a template class of *PathExecutionState* and it can be used to derive statistics on the execution of the Path (e.g., a simple collection of the walker position and of the physical parameter upon each sampling time which can be analysed after the execution to

infer such parameters as the average or the peak velocity). The specific algorithm implemented will be developed in WP6. Relevant methods of *GuidanceAlgorithm* are

- *activateGuidance:* used to pass a new path to be executed to the guidance algorithm;
- *start:* used to initiate a new *Path;*
- *stop:* used to halt the execution of a *Path;*
- *waitForNextActivation*: suspends the execution of the task animating the object for an interval of time given by the Period (which is a template parameter associated with the object).
- *addObserver:* used to register a new observer
- *getPathState:* used to provide a snapshot of the state of the path execution.

The *ReactivePlanner* is a class to generate a new plan when the conditions specified in *UserPreference* cannot be met using the current plan. Just as the *GuidanceAlgorithm* it is an active object operated by a real—time task that is activated once every *Period* (where *Period* is a template parameter of the class). The *ReactivePlanner* samples its sensors, consults the current position on the Path and decides if the evolution is feasible (i.e., compatible with the *UserPreference*). Sensors can convey information from the environment and from the user. The specific algorithm implemented will be developed in WP5. It is possible to connect a *TaskObserver* to the planner to keep track of all events that could be potentially be noted in the execution of a Task (e.g., the number of occlusions found along the way, possible changes in the physical of the emotional state of the user). Relevant method of *ReactivePlanner* are:

- *activatePlanner:* used by the *ActivityExecutionEngine* to initiate a new task
- *start:* used to start the execution of a task
- *stop:* used to halt the execution of a task
- *getTaskState:* returns the current state in the execution of a task
- *addObserver:* registers an observer to monitor the execution of the task
- *waitForNextActivation:* the execution is suspended to wait for the next periodic activation
- *checkPath:* used to check if a Path is still feasible given the sensor readings.
- *replan:* to compute a new plan if the check of the path feasibility fails.

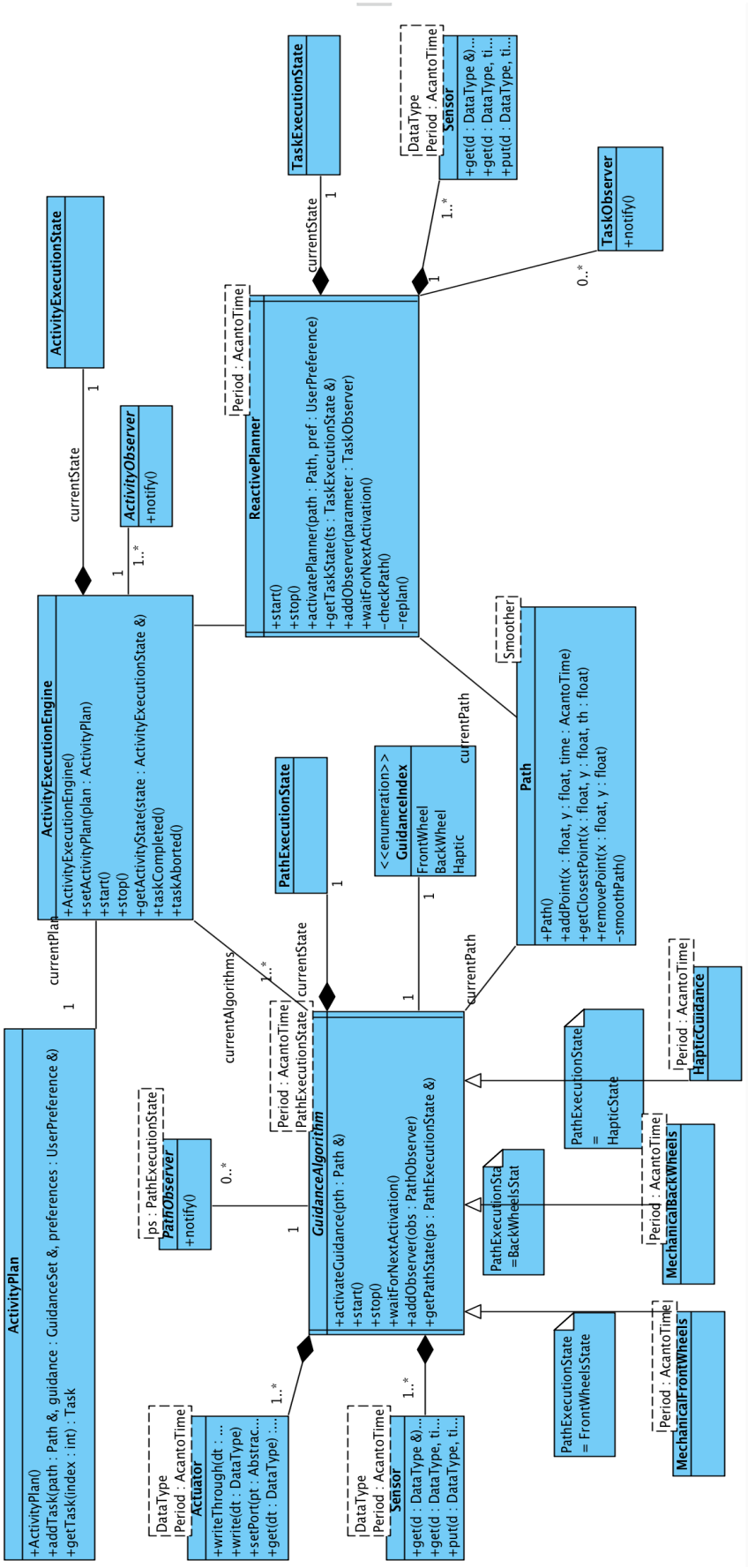

Figure 4: The Execution Engine

A possible interaction between the different entities is shown in the sequence chart in Figure 6. We assume the presence of a single Observer (in fact we could have as many as needed). The diagram is explained as follows:

1. The launcher (that we do not specify here) initiates an activity by invoking *setActivityPlan* on the instance *Engine* of *ActivityExecutionEngine*
2. The launcher calls *start* on *Engine*

   2.1. Engine *notifies* the start event to the observer that samples and notes the state (2.1.1)

3. *Engine* extracts the first task from the *ActivityPlan*
4. *Engine* activates the instance *guidance of GuidanceAlgorithm* by passing it a reference to the *Path*
5. *Engine* activates the instance *rp* of *ReactivePlanner* passing it a reference to the *Path* and to the *UserPreference*
6. *Engine* starts *guidance*
7. *Engine* starts *rp*

   7.1 rp monitors the execution of the task, until it detects that the Path has been completed. When this event occurs *rp* calls *taskCompleted* on *Engine*

   7.1.1 *Engine* stops *guidance*

   7.1.2 *Engine* stops *rp*

   7.1.3 *Engine* notifies the end to the ActivityObserver, which samples the state and notes it

8. *Engine* extracts the next task in the activity and the same sequence as abose is re-iterated.

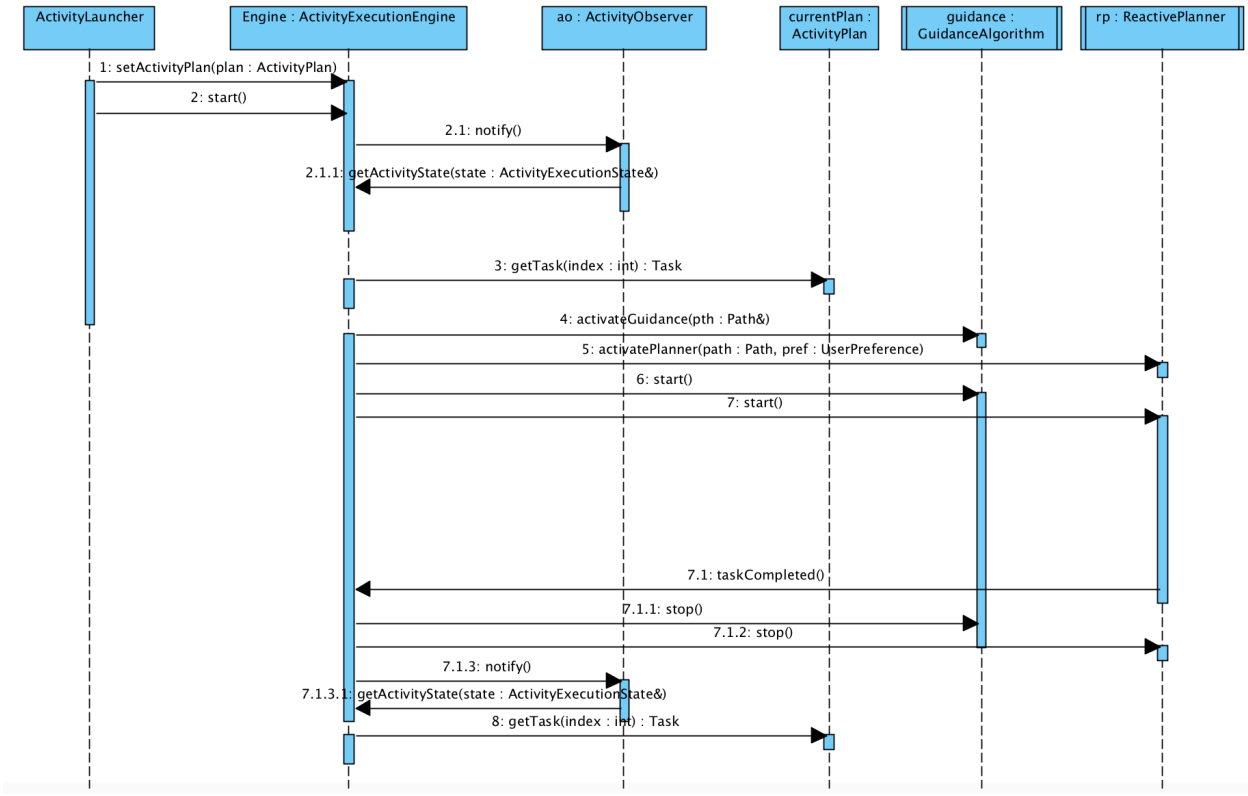

Figure 5: sequence chart of a task's execution

3.3.3 Example execution of the GuidanceAlgorithm

Looking at the sequence chart in Figure 6, it is now worth focusing on the sequence of operations performed by the *guidance* between step 7 (start) and 7.1.1 (stop).

This is illustrated in the sequence chart in Figure 7. We assume that the guidance algorithm uses the motorised front wheels to steer the walker and that it only requires to the position of the

walker with respect to the path. Furthermore we assume the presence of a single observer (we could easily generalise to an arbitrary number of observers using the Observer Design Pattern[2]).

The steps of the sequence diagram are as follows:

1. *Engine* starts *guidance*
   1.1 *guidance* notifies the start to the *obs (*an instance of *PathObserver). obs* samples the state and notes the event
2. *guidance* blocks on a *waitForNextActivation* call
3. When the Period expires *guidance* wakes up and samples the localisation sensor (*localisation)* through a *get* call
4. *guidance* queries *currentPath* for the point on the *Path* which is the closest to the *walker*
5. *guidance* computes the control action and sends it to the actuator through a *writeThrough* call (see later).
6. *guidance* calls the *notify* method on the *observer (*which samples the *guidance* state).
7. *Guidance* blocks again on a *waitForNextActivation* cal and the cycle is repeated.

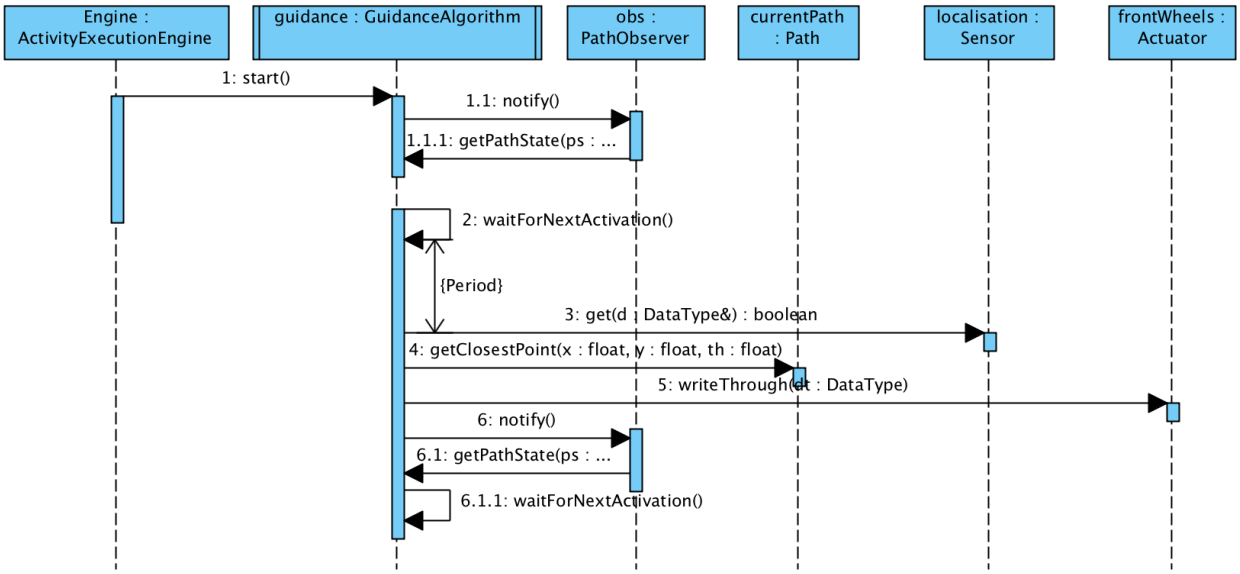

Figure 6: Close-up on the execution of the GuidanceAlgorithm.

3.3.4 Example execution of the ReactivePlanner

We now show three different execution scenarios for the ReactivePlanner.

We assume that our *ReactivePlanner* only samples the surrounding environment (through an appropriate instance *EvironmentSensor* of a *Sensor*), and that it is attached to a single instance of *TaskObserver* named *to.*

In the first scenario (reported in Figure 8) we show a “nominal” execution where no exceptional event occurs that could determine changes in the Path.

The sequence diagram can be described as follows.

1. *Engine* starts the *ReactivePlanner rp.*
   1.1 The event is notified to the *to TaskObserver*
      1.1.1 *to* samples the *rp* state
2. *rp* blocks awaiting the next periodic activation
3. when *rp* is awakened it samples the localisation sensor to find the current state
4. *rp* sample *environmentSensor* to identify obtacles in the surrounding, as well as the presence of other human and of the partners of the user.

5. *rp* queries *currentPath* to identify the position of the walker on the planned path
6. *rp* calls its own *checkPath* method to se if any violation of the constraint is foreseeable in a future time horizon, given the current situation. The planned path is still feasible.
7. *rp* notifies the event to the *to TaskObserver*
   7.1 *to* samples and notes the *rp* state
8. *rp* blocks again until the next activation, where the cycle is repeated.

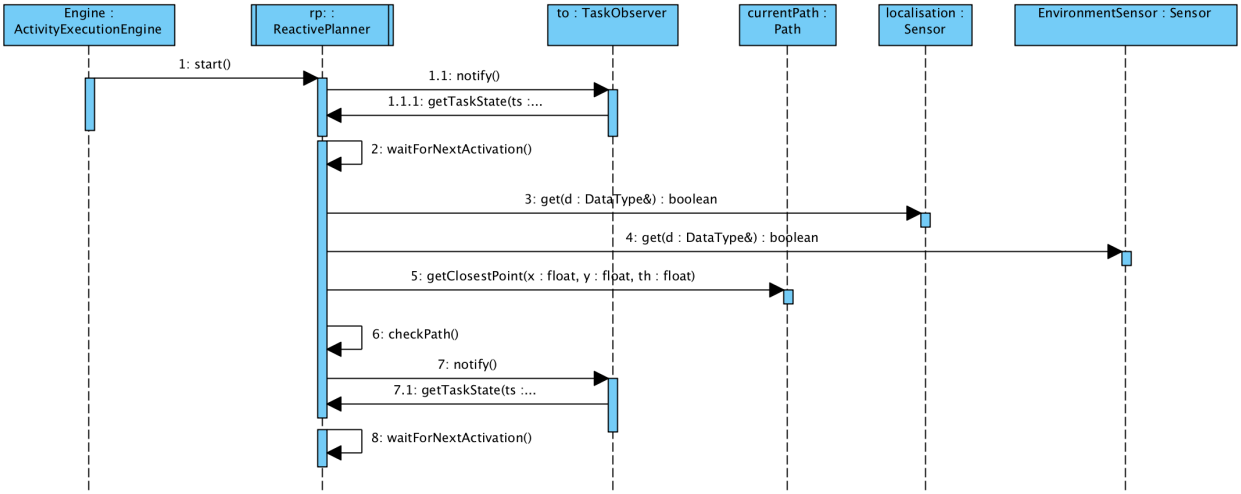

Figure 7: ReactivePlanner in action the nominal execution case

In the next diagram (Figure 9), we show how to manage “exceptional” situations in which the device encounters obstacles or exceptional situations. Up until step 6 the Diagram goes as in the nominal case depicted in Figure 8. We start describing the from this step:

6. *rp* calls *checkPath()* but the check fails meaning that the Path is infeasible
7. *rp* seeks a new plan complying with the constraints
   7.1 the call to *replan* determines the deletion of a point of the current path
      7.1.1 after the deletion of a point *smoothPath* smoothens the *Path* generating intermediate points in between the waypoints.
   7.2 a new point is inserted to replace the removed one
      7.2.1 *smoothPath* is called once again
8. *rp* notifies the replanning event to the *to TaskObserver*
   8.1 *to* samples and notes the state
9. *rp* blocks awaiting the next periodic activation

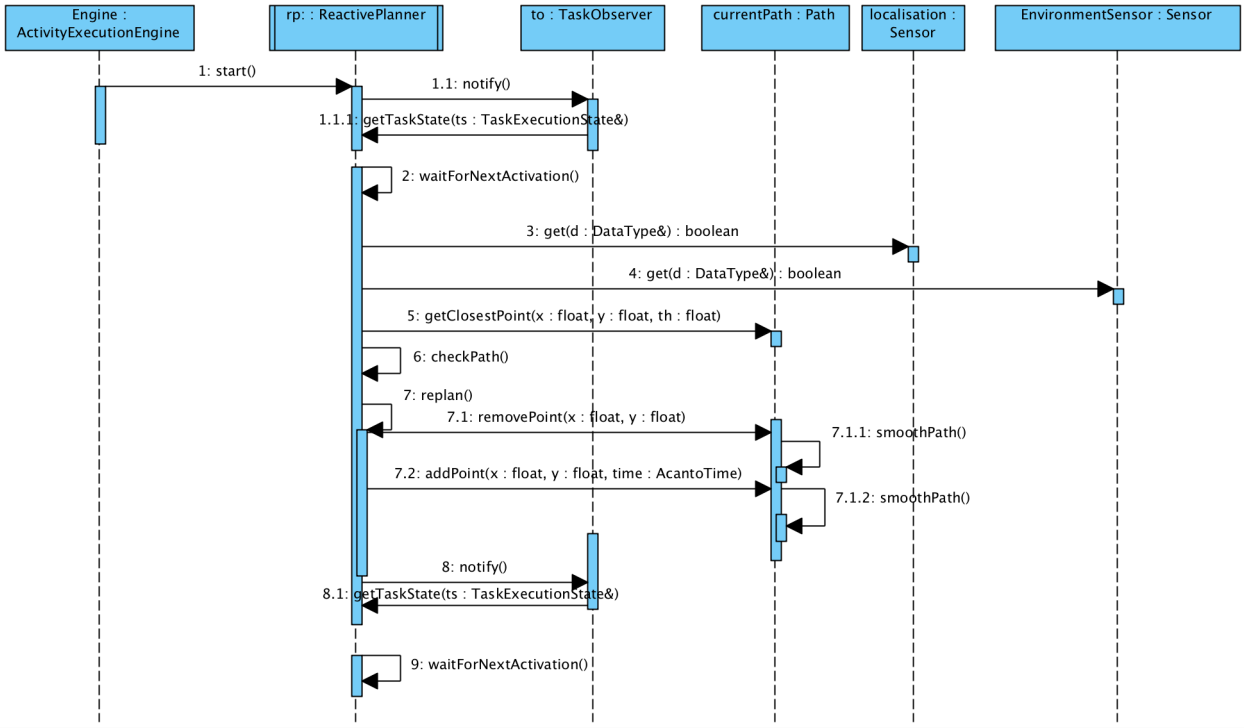

Figure 8: the ReactivePlanner in action: the case of successful re-planning

We finally document a case of unsuccessful re-planning in the sequence chart in Figure 10. The chart goes as the case of Figure 9 up until step 7. So we start discussing from this step on.

7. the call to *replan* fails because no feasible alternative plan can be found
8. the event is notified to the *to TaskObserver*
9. as a consequence of the failure, *taskAborted* is called on *Engine*
    9.1. *rp* is stopped (and so is guidance). The necessary book-keeping is made (i.e., calls to the observer)
    9.2 *replan* is called on *ActivityPlanner*

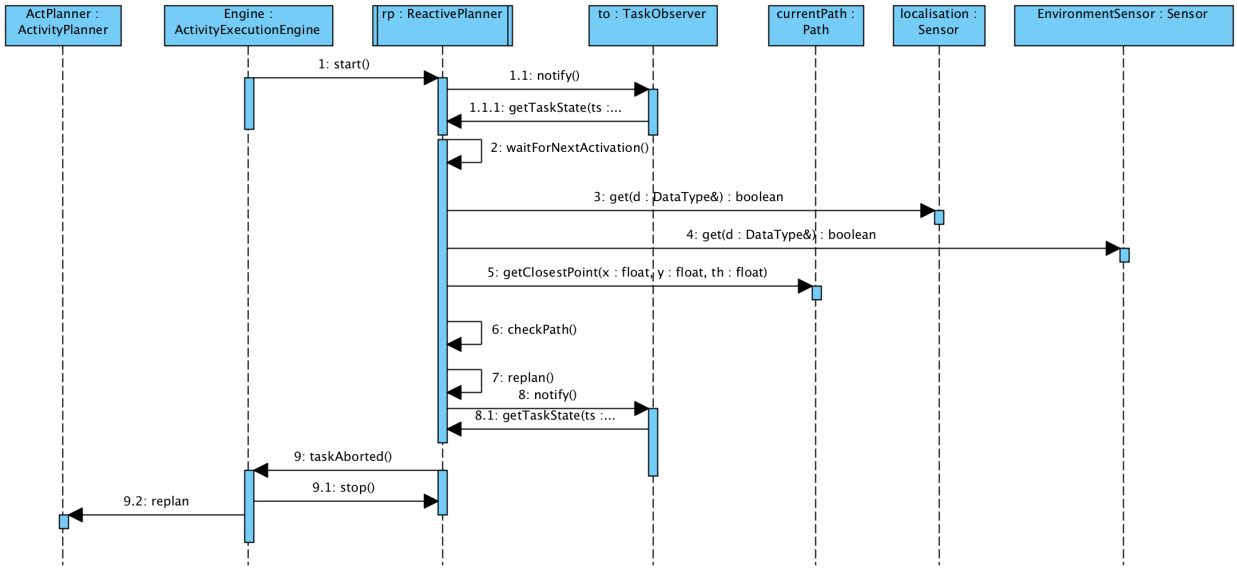

Figure 9: ReactivePlanner in action: the case of unsuccessful re-planning

3.3.5 The Hardware abstraction layer

In the illustration of guidance algorithms and of reactive planning presented above, two classes play a key role: *Sensor* and *Actuator*. Such classes are actually an abstraction layer to the hardware devices embedded in the walker and to the various sensing devices used by its cognitive component (i.e., for the detection of environment and of user emotional and physical state). A key architectural choice inherited from the DALi project is the use of the zeroMQ middleware to interconnect the different components. In essence, each sensing component embeds its data in a zeroMQ packet, which is received and processed by one of the computing elements deployed on the FriWalk.

The specification of the different zeroMQ packets is, at the moment of this writing, under development as newer and newer components become available for the integration.

In this section we focus on what happens after a packet from a "physical sensor" is received or right before a new packet is composed and sent to an actuator. This is what we refer to as "hardware abstraction layer" (HAL).

The class diagram of the HAL is depicted in Figure 11.

The first class we introduce is *Sensor*, which is a template class. Its parameters are the *DataType* carried by the *Sensor* and the *Period* the sensor is updated with. Its main methods are:

- *get:* is used by the "owner" of the Sensor to retrieve its contained data. We have got to versions of *get*. The first one is asynchronous. The methods returns a piece of data (instance of the *DataType* template parameter) and a Boolean, which is true if the data are fresh or false if the same data have been used already. The second version of *get* blocks the caller for a specified amount of time. In this case, the method returns the data and a Boolean flag, which is set to true when the *get* call results in an expiration of the timeout.
- *put:* is called whenever a new element of data is available from the zeroMQ middleware. The *put* call requires the data element and the timestamp recorded when the data are fetched. The *put* call is ignored and the data are discarded if it occurs before the expiration of the next time period.

Sensor objects have to be connected to a *SensorPort,* which is a class directly interfaced with the middleware. More sensors can be connected to the same port and used by different applications. *SensorPort* is a template class of the *DataType* carried and of a *PortDescriptor* and is implemented as a singleton, i.e., a class that can have only a single instance[i]. We omit the technicalities here to implement a singleton because they are common knowledge. *PortDescriptor* is a data—type containing a number of compile—time parameters such as a port identifier and the *Period.*

An example definition for the PortDescriptor is as follows:

```
///     typedef struct P1 {
///        typdef int port_id_type; /* type for port identifier */
///        static const port_id = 1; /* identifier of a port */
///        typedef PeriodXXX Period;
///     } Port1;
///
```

where the Period can be defined as

```
///@brief Aperiodic
typedef AcantoCommon::AcantoTimeConstant<int64_t, 0,AcantoCommon::Granularity::MILLI>
APERIODIC;

///@brief Localisation period
typedef AcantoCommon::AcantoTimeConstant<int64_t, 4, AcantoCommon::Granularity::MILLI>
PeriodLoc;
```

The use of static type definitions enables the enforcement of compile time checks in the definition of sensors and ports that prevents errors in the intended semantics of the objects without *paying* the execution of run—time checks. In this case, we can enforce that the connection of a *Sensor*

to a *SensorPort* can be made only if both of them carry data of the same type and have the same *Period*. *Sensor* can have a subsampling parameter. If a *Sensor* is connected to a *SensortPort* that is created with subsampling parameter 2, it means that it collects one piece of data from the port out of two, Relevant methods for *SensorPort* are

- *bind*: used to bind a sensor
- *disconnect*: used to disconnect a *Sensor* for the *SensorPort*
- *put:* called by the middleware clients to insert a data element into the port, which is then propagated to all *Sensor* objects connected to the port.
- *fire:* is an internal (private method) used to insert the newly arrived data into all *Sensor* objects connected to the Port.

The structure for the actuators is to a large extent similar to the one for sensors. Once again, two classes are the cornerstones: *Actuator* and *AcutatorPort,* the former being the programming abstraction used by the guidance applications and the latter being the primary way to interconnect to the middleware and insert the data into the "actual" actuator. The first important difference with *Sensor* is that only one single *Actuator* instance is allowed to connect to a port at a given time. The most important methods of *Actuator* are:

- *writeThrough:* is used to insert a data into the actuator and send it through the middleware straight away.
- *write:* this is used to implement a time—triggered semantics[3] whereby data are written into the actuators only when the period expires. Therefore, a periodic write is not immediately actuated but is deferred to the sampling period expiration. This method is enabled only id the *Period* is parameter is not null. Otherwise it is compiled out using the *enable_if* c++ 14 construct.
- *get:* is used only if the period is not null by the middleware to fetch the data and pass it over to the actuators.
- *setPort:* is used to set a pointer to an *AbstractDataSink* interface (which the ports shall implement) to direct the write to.

The *AbstractSink* interface is used to allow the use of *ActuatorPort* by Actuator avoiding the problem of cyclic dependencies. Its only method is

- *put:* used to insert a data element into the actuator port.

Very much like SensorPort, *ActuatorPort* is once again a singleton (we omit the details for the implementation of the pattern) and is a template of the *PortDescriptor* and of the *DataType* carried. The only important difference is that in order differentiate the behaviour of *periodic* and a-periodic ports it was in this case to create a hierarchy, the use of *enable*_if for some methods being insufficient. In this case the difference between the two semantics is not limited to the presence or absence of some methods, but is more radical. Indeed, the time-triggered semantics of the periodic case requires that the port be an active object, which wakes up on period expiration to fetch the data from the *Actuator* bound and pushes it through the middleware. In the Aperiodic case, on the contrary, the port can very well be a passive object.

Ports use an interface (*AbstractUpdatePortManager*) which is an interconnection to the port middleware. The key method is

- *update:* used by the port manager to retrieve the data from each actuator port and send it to the middleware.

The hierarchy for the *ActuatorPort* is rooted into the class *BaseActuator* port that implements the following methods:

- *put*: inherited from AbstractDataSink with the same semantics explained above
- *bind:* used to interconnect a new actuator. If an actuator is bound already, the call disconnects it.
- *disconnect:* is used to disconnect an actuator from the port.
- *start:* it starts the operation of a port. It is of practical use only if the port is periodic.
- *stop*: it stops the operation of a port.
- *get*: it is used by the middleware client to retrieve the data element stored in the port as a result of a call to the *update* method. It returns the data element and a

Boolean set to true if new data are store in the port that have not been fetched before and false otherwise.

The specialisation of ActuatorPort for the a-periodic case does not have much to say. More interesting is the periodic case. In this case the port is connected to a *PortThread*, which essentially turns the periodic ActuatorPort into an ActuatorPort.

PortThread is a template of the activation Period and of the Port it animates. Its most important methods are:

- *start:* used to start the thread
- *stop:* used to stop the thread
- *body:* it is executed on every periodic activation and essentially calls the *updateTrigger* on the port causing the data update.
- *waitForNextActivation:* used to block the task waiting for the next activation

The periodic specialisation of *ActuatorPort* has the following methods:

- *start*: used to start the thread managing the port
- *stop*: used to stop the thread
- *updateTrigger* : called upon each periodic activation to retrieve the data from the port.

The *WalkerDeviceManager* class mediates the interconnection between the ports and the middleware. It encapsulates and manages all the different ports. To this end it exploits four different helper classes: two for the management of the localisation system (*RemoteLocalisationClient*, *RemoteLocalisationSubscriber*) and two for the management of the mechatronic subsystem (*WalkerRemoteClient, WalkerSubscriber*). Other similar classes will be defined with the same interface for the integration of environment and user sensing. The class has the following methods:

- *init*: used to initialise the ports (e.g., setting the sampling periods).
- *start*: used to start all the operations of the instances of the helper classes (rlc for *RemoteLocalisationClient*, *wrc* for *WalkerRemoteClient, rls* for *RemoteLocalisationSubscriber, rlc* for *RemoteLocalisationClient)*.
- *stop*: used to stop the operations of the four helper objects.
- *linkSensor*: used to link a *Sensor* to a *SensorPort.* The method is a template of the port type. This secures that connections are properly made at compile time.
- *unlinkSensor*: used to remove a sensor
- *linkActuator*: used to link an Actuator to an ActuatorPort (identified through its template parameters).
- *unlinkActuator*: used to remove an actuator.
- *writeLocPorts*: function called by the loc_callback to insert a newly arrived localisation packet into the port
- *writeHwPorts*: called to insert a newly arrived data element from the sensing subsystem into the port
- *ls_callback*: callback function called every time a new data arrives from the walker sensor connected to the zeroMQ middleware
- *loc_callback*: callback function called every time a data element related to the localisation subsystems arrives through the zeroMQ channel.

We omit for the sake of brevity the description of the helper classes. It is more interesting to look more closely at the sequence charts implementing the data exchange between middleware, *Sensor, Actuator, SensorPorts and ActuatorPorts*.

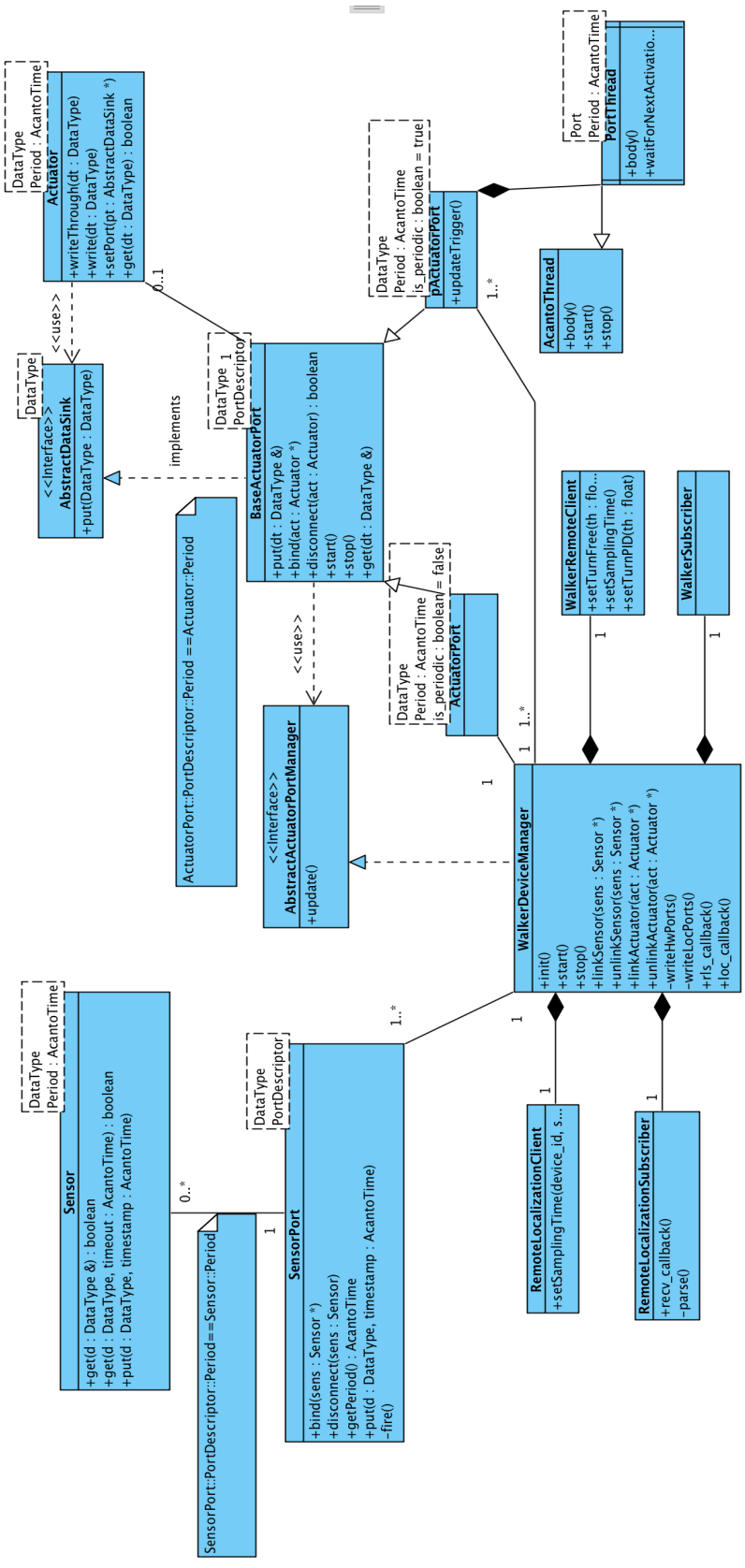

Figure 10: class diagram of the HAL

The first sequence chart (in Figure 12) shows the management of *Sensor* data. The exchanged messages are as follows:

1. the startup code calls *init* on the *dm* instance of the *WalkerDeviceManager*

1.1 *dm* calls *setSamplingTime* on *rlc* to set the sampling time of the localisation system.
1.1.1 *rlc* sets up a proper Json string and transmits it over the zMQ bus.

2. *Engine:* activates the guidance *GuidanceAlgorithm.*
2.1 *guidance* links its sensor to the *SensorPort* for localisation by calling linkSensor on *dm*
2.1.1 *dm* calls *bind* on the SensorPort

3. after a *Period* duration a first sample arrives in the middleware triggering the recv_*callback* execution
3.1 recv_callback parses the packet
3.2 loc_*callback* is called on dm
3.2.1 loc_*callback* calls *writeLocPorts,* which prepares the appropriate data structures to insert the data into the port
3.2.1.1 *put* is called on the localisation port
3.2.1.1.1 *put* calls *fire*
3.2.1.1.2 *fire* calls *put* on all the sensors connected to the port

4. in a separate thread of execution *guidance* samples the sensors, which receives the freshest data.

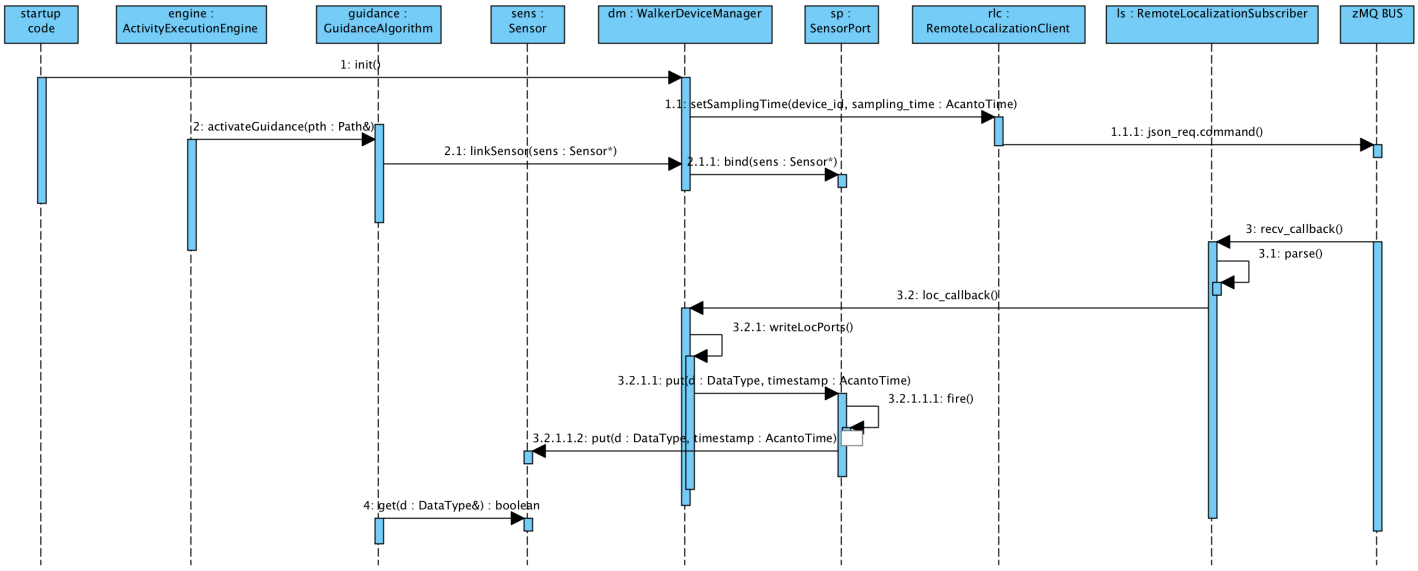

Figure 11: sequence chart for the insertion of data into the sensors

Now we use Figure 13 to show a sequence chart for the use of an actuator in the writeThrough modality. The chart goes through the following steps

1. *ActivateGuidance* is called on *guidance*
1.1 guidance calls *linkActuator on dm*
1.1.1 dm calls *bind* on the *ActuatorPort*

2. At some point in its execution *guidance* calls *writeThrough* on the *Actuator*
2.1 calls *put* on the *ActuatorPort*
2.1.1 put calls update on *dm*
2.1.1.1 update queries in sequence the different ActuatorPort until it
finds the one responsible for the request (which return true to get)
2.1.1.2 *update* calls a method on the walkerClient to turn the wheels
to a specified position
2.1.1.2.1 the *setTurn* generates the packet and sends it over

through the middleware

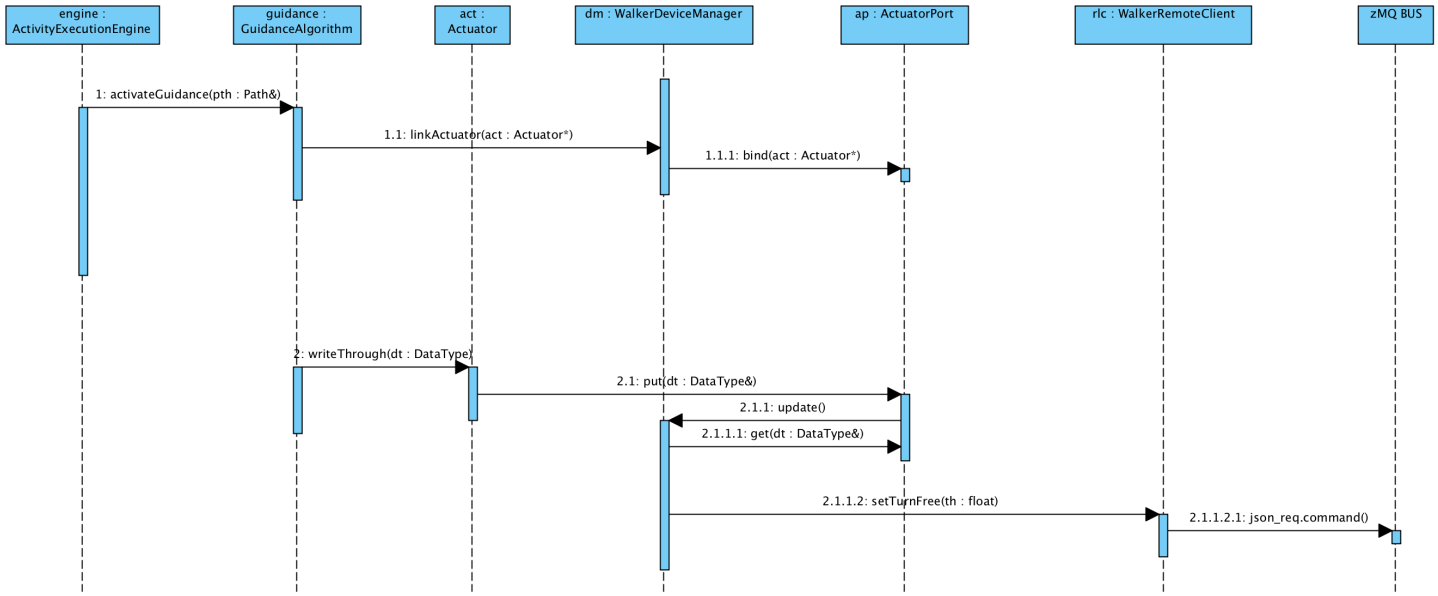

Figure 12: insertion of data into an Actuator using the writeThrough modality

Finally, we look at the same type of interaction using a time—triggered Actuator. The sequence chart is shown in Figure 14.

1. The first step is as for the writeThrough case. *Engine* activates guidance
   1.1 *Guidance* calls *linkActuator* on *dm*
      1.1.1 *linkActuator* calls bind on the ap *ActuatorPort*
         1.1.1.1 *bind* calls start on *ap*
         1.1.1.2 the call to start on *ap* determines a start call on the PortThread
            1.1.1.2.1 The thread blocks until the next period
2. *guidance* calls *write* on the Actuator. Contrary to what we discussed in Figure 12, the write is not immediately propagated to the port, and the data are stored locally within the actuator.
3. After a Period expires, the *PortThread* awakens and calls *updateTrigger* on the ap *ActuatorPort*
   3.1 *updateTrigger* retrieves the data from the Actuator *through a* get call
   3.2 *updateTrigger* inserts the received data into the port through a *put* call
   3.2.1 put calls *update* on *dm* and from this point on everything proceeds as in the writeThrough scenario in Figure 12.

Chapter 4 - Perception of users and environment

Our design choice based on the adoption of zeroMQ middleware allows us to decouple the development of the execution components (documented in chapter 3) with the sensing components.

On the producer side, the only thing that we need to specify for each new component integrates is: a “topic” (in essence a packet type), the content of each message related to the topic, and the period at which such messages are sent to the middleware. The zeroMQ middleware allows us to abstract from the physical deployment of the different sensing components on the walker hardware and from the networking protocol used. All we need to do is to define the syntax of each packet and use the JSON based api offered by zeroMQ to stream the data. The specific syntax of the packets is currently under definition and will be subject to changes and adaptation, as demanded by the different components developed during the project.

On the consumer side, the way environment and user data are made available to the application is by extending the HAL described in the previous chapter with the addition of

- new client and subscriber classes for each new component to be integrated (e.g., EnvironmentClient, EnvironmentStreamer): the former is used to control the component, the latter to receive, decode the messages and call the appropriate callbacks
- insert new callback function on the device manager
- define SensorPort to support the new data.

After these steps are taken, the design is perfectly compatible with the execution environment described in the previous chapter.

Bibliography

[1] ACNTO Consortium. ACANTO Project, Feb. 2015.

[2] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides John (1995), Design Patterns: elements of Reusable Object-Ortiented Software, Addison-Wesley. ISBN 0-201-633361-2

[3] Kopetz, Hermann, and Günther Bauer. "The time-triggered architecture." Proceedings of the IEEE 91.1 (2003): 112-126.