Mobile Robot Localisation and World Modeling in a Real-Time Software Architecture

Tesi di dottorato di
Daniela Laura Micucci

Dottorato di Ricerca in
Matematica, Statistica, Scienze Computazionali e Informatica

Università degli Studi di Milano
Dipartimento di Matematica "Federigo Enriques"

XVI Ciclo

Ph.D. Programme Coordinator: Prof. Giovanni Naldi

Advisor: Prof. Francesco Tisato

Co-Advisor: Dr. Fabio Marchese

"*Mum, Dad: don't worry, I'm still on this planet!*
I will come back to spend a little more time with you"

Abstract

The goal of autonomous mobile robotics is to build physical systems that can interact with environments not specifically structured for this purpose.

Even if the applications that might exploit autonomous mobile robots are widespread, current technologies are still immature at satisfying the growing requests. For this reason, robot navigation constitutes one of the major trends in the current research on robotics.

A precondition for a mobile robot to be autonomous is the ability to self-localise inside an environment. This precondition is difficult to satisfy when the robot does not exploit a map of the environment to localise itself.

Current research investigates methods for map learning, based on the detection of natural features. These methods should allow a robot to self-localise inside the environment it is exploring, and contemporarily to build an incremental representation of the same environment.

Research on these methods is still in progress. This is due to the fact that the problem they face is hard because of the following *paradox*: position estimation needs a model of the environment, and world modelling needs the robot position. "Which come first, the chicken or the egg?"

Current research answers the question by proposing solution based upon the simultaneity of the activities. These kinds of approach are known under the SLAM (Simultaneous Localisation And Mapping) acronym, and support the idea that the two activities should be performed together.

The aim of this work is to propose a novel approach based upon the *concurrence* of the two activities. This approach, named CLAM (Concurrent Localisation And Mapping), is founded upon the conjecture that a proper separation of concerns may help in breaking the loop of the "chicken and egg" problem. Localisation and Modelling, acting on different time scales, are mostly independent each other. Sometimes a synchronisation is needed, but controlled by an external and suitable strategy.

We consider the CLAM system a *time-sensitive* one since it has to perform a number of different activities with multiple, dynamic, and interdependent temporal requirements. Furthermore, a CLAM system must be able to dynamically change the activities temporal requirements.

To fulfill the goal we have to define and implement a general framework for the construction of time-sensitive systems.

The framework, named Real-Time Performers, is composed by a reference architecture and a working implementation providing software engineers a consistent set of software modules to build time-sensitive systems. The architecture is based upon a novel methodology based on computational reflection, this methodology models the temporal behaviour of the computational system with a set of suitable architectural abstractions, reifying time related aspects of the system itself.

The final result of this work consists in a real implementation of a system supporting the exploration activity of a robot equipped with an odometric system (for positioning) and a trinocular stereo system (for environment perception). CLAM principles and Real-Time Performers architecture have driven the design of this system. Finally, the resulting system has been developed exploiting Real-Time Performers framework.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Robotics is a branch of engineering that involves the conception, the design, the manufacture, and the operation of robots.

A *robot* is a complex system designed to execute with precision one or more tasks. There are as many different types of robots as there are tasks for them to perform.

Robots may be classified exploiting different criteria: their *generation* and their *level of intelligence* are the most used since are exhaustive. Concerning the level of intelligence, robots mainly fall into either of three categories: autonomous robots, fleets of insect robots, and flocks of robots. An autonomous robot acts as a stand-alone system, complete with its own computer (called the controller). Fleets of insect robots are robot working in fleet, where single robot has its own controller[1]. Flocks of robots are similar to fleets of robot, but the members work under the supervision of a single controller.

We will focus on a particular kind of autonomous robots: robots operating inside structured environments. An autonomous mobile robot should be self-contained, i.e., it should have its own power supply, electro-mechanics, sensors, and actuators for moving and performing tasks, processor systems for perception, computing and control, and devices for communicating with the outside world.

The development of techniques for autonomous robot navigation constitutes one of the major trends in the current research on robotics.

This trend is motivated by the current gap between the available technology and the demands of new applications. On one hand, current industrial robots lack flexibility and autonomy: typically, these robots perform pre-programmed sequences of operations in highly, constrained environments,

[1]The term insect arises from the similarity of the system to a colony of insects, where the individuals are simple but the fleet as a whole can be sophisticated.

and are not able to operate in unknown environments or to face unexpected situations. On the other hand, there is a clear emerging market for real autonomous robots. Possible applications include: intelligent service robots for offices, hospitals, and factories, domestic robots for cleaning or entertainment, semi-autonomous vehicles for helping disabled people, surveillance, rescue, and so on.

Fundamental components of any mobile robot systems are the *localisation and navigation systems* [10]. Leonard and Durrant-Whyte [64] summarised the navigation problem as follows: to autonomously navigate, a robot should be able to answer the following three questions: *"Where am I?"*, *"Where am I going?"*, and *"How should I get there?."* The first question is termed *localisation*, while the second and the third deal with goal-planning and path-planning respectively.

A robust and reliable solution to the localisation problem is, in any case, a precursor to goal-planning and path-planning activities [64]: if a robot is not able to localise itself inside an environment, all the more reason it is not able to plan a path achieving the desired goal.

In industrial settings, the localisation problem may be solved by the exploitation of artificial markers helping the robot in its self-localisation process. Where markers are not available (or not installable), then it is possible to provide the robot with a preloaded static map of the environment it will navigate in.

This kind of solution cannot be applied for robots operating in office or domestic environments. It might not be acceptable to place artificial markers in these environments. It is in general difficult to assume the presence of a map describing the environment geometry since the production of a map is a long work requiring specific competencies.

Consequently, there is a great interest in investigating methods allowing map learning (i.e., automatic environment maps reconstruction), based on the detection of natural features, to be used for robot self-localisation and navigation.

The aim of these methods is to make a mobile robot able to autonomously localise inside an unknown environment. To fulfill this requirement, the robot should be able to self-localise inside the environment it is exploring, and contemporarily to build an incremental representation of the same environment.

The self-localisation and the world modelling activities are *per se* quite simple to be performed separately, if respectively an accurate world representation is available and the robot position inside the environment is accurate. When the two activities have to be performed together without neither the map, nor the exact position, the problem increases in difficulty. Moreover, the imprecision (e.g., noises, errors, vibration, and so on) of the perceptive systems of a mobile robot yields to more difficult issues.

The complexity of the problem increases also for the strict relationship between the two activities: position estimation needs a world model, and world modelling needs the robot position. How to break these crossed dependencies between these two information gathering processing? Leonard and Durrant-Whyte in [65] say “This problem is difficult because of the following paradox: to move precisely, a mobile robot must have an accurate environment map; however, to build an accurate map, the mobile robot's sensing locations must be known precisely [. . .] *which came first, the chicken or the egg?*”

1.2 Motivations

Current approaches address the “chicken and egg” problem by exploiting solutions based on the *simultaneity* of the two activities. This kind of approaches are known under the SLAM acronym that stands for *Simultaneous Localisation And Mapping* [65]. SLAM approaches are based upon the idea that a vehicle (to operate in autonomy inside an unknown environment with no initial position information) should be able to build a map of its environment while simultaneously uses that map to self-localise.

Approaches based on SLAM philosophy perform in any conditions both the activities. This operating mode may be useless when the differences between the outputs produced at each execution by one of the activity are trivial. This is the case of modelling activity. Indeed, the partial reconstructed map of the environment (world model in the sequel) may be often considered *valid* for a long period. A valid world model is a map containing all the necessary information for a successful robot localisation.

The long period validity of the world model, is also enforced by the fact that when a robot explores an unknown environment, its displacements will be small enough to allow the robot not to lose itself. This means that the real world that it perceives at each movement will not substantially change.

For these reasons, updating the world model does not bring any more information for robot self-localisation. What is worse is that a lot of time and resources are unnecessarily wasted.

Simultaneity not also implies that the activities cannot be executed separately, but also that it is impossible to assign them different timings. This is an disadvantage when, for any reasons, one activity must be executed asap.

From our point of view, in fact, SLAM target systems are time-sensitive since their execution is subjected to temporal constraints. It is fundamental for these systems that the right operations are performed at the right time. All the more reasons, when anomalies occur. Suppose that a robot, for any reason, does not localise itself. It is reasonable to think that the first (in a temporal sense) operation to perform is to decrease its speed. This is only one example, but what emerges is that time plays a crucial role in those kind of systems. Consequently, the ability in managing both timing and correct temporal execution of the activities are key issues.

From a software engineering point of view, what is also missed in current solutions based on SLAM is an *architectural* approach to the problem.

SLAM approaches often neglect an overall architectural organisation of the software system, focusing only on its algorithmic aspects.

The result is, in general, a software in which functionalities are merged and mixed one another. But, what is worse, software so structured, poorly adapts itself in a dynamical way to the different situations that may occur. Concerning time-sensitive systems like SLAM target ones, the adaptivity of activities execution *timing* it is a key point to achieve the temporal correctness of the software system when exception occur.

In the software life-cycle, the algorithms definition is made after the analysis and design phases whose accuracy are decisive for a successful software. A good system design leads to define components and to build the system by their composition [102]. If the definition of components do not embed any kind of strategy, then a lot of advantages may be achieved, e.g., the same components may be used under different strategy, and the components may be easily substituted with more efficient ones. Moreover, keeping in mind computational [104] and architectural [16] reflection concepts, when designing a system, then its execution may be easily observed and opportunely controlled exploiting suitable strategies. The identification of the right *architectural abstraction*s leads to realise software systems that may adapt themselves to any kind of circumstance may occur during their execution.

1.3 Contributions

The contribution of this thesis can be described as follows:

1. We propose a new approach to "Localisation and Modelling" based upon the following key concepts:

   - *Localisation* and *Modelling*, both relying on *Perception*, are the basic activities performed by a robot when exploring an unknown environment;
   - *Localisation* and *Modelling* operate on separate information and are subject to different timing constraints. Therefore they can be performed concurrently and with independent timings;
   - Localisation relies on information which loosely depends on the information generated by *Modelling*, and vice-versa. Therefore *Localisation* and *Modelling* must synchronise whenever a criticality arises, i.e., whenever the information an activity relies on is not reliable;

- Synchronisation is controlled by a strategy which relies on the observation of the criticalities and drives the relative rates of the activities. This is made easier by exploiting the fact that the physical speed of the robot can be controlled.

We named this approach *CLAM*, which stands for *Concurrent Localisation And Mapping.* Even if SLAM acronym differs only in one character with the one we propose, this character makes the difference. The substantial difference is the modality with which the two activities may be executed. SLAM treats the Localisation and Modelling *simultaneously*, CLAM *concurrently.* Basic conjecture of this thesis is that a proper separation of concerns should help breaking the chicken-and-egg-loop. Even if Localisation and Modelling are related, they act on different time scales, so that they can be considered as mostly independent activities which sometimes synchronise under the control of a suitable strategy.

2. Since current approaches to the design of time-sensitive systems do not address temporal aspects of computation at the architectural level, we propose, also, a software architecture named *Real-Time Performers* that reifies a set of architectural abstractions that properly capture the temporal behaviour of the system. This software architecture may be considered a reference software architecture providing the basis upon which real-time systems may be designed. Within the software architecture, we have designed and implemented a general *framework.*

To test the validity of both the CLAM approach and the Real-Time Performers architecture (and framework), a real implementation has been realised to support the exploration activity of a robot equipped with an odometric system (for positioning) and a trinocular stereo system (for environment perception).

To this aim, we have designed our CLAM system using Real-Time Performers building blocks. Successively, we have implemented a running system exploiting the Real-Time Performers framework. The specific algorithms coding the single steps of robot localisation have been developed as simple as possible. This is justified by the fact that the aim was not the invention of new algorithmic solution, but the creation of a new architectural approach. As a matter of fact, given a sound underlying architecture, the substitution of the trivial implementation of each algorithms with more clever solutions is possible without affecting the overall system.

1.4 Outline of the Thesis

In the present chapter has been given a brief introduction about both the problem we have faced and the motivations of our work. The chapter concluded providing an overview about the topics covered in this thesis.

The rest of the thesis is organised as follows:

- Chapter 2 deals with a brief overview about the state-of-the-art concerning the robotics area dealing with autonomous mobile exploration. The chapter ends with an overview about software architecture and reflection mechanisms;
- Chapter 3 presents CLAM approach in detail justifying the idea behind it;
- Chapter 4 presents Real-Time Performers reference architecture focusing on its main principles;
- Chapter 5 covers the Real-Time Performers framework, describing its main concepts and usage;
- Chapter 6 presents the system that we realised according to CLAM principles and designed exploiting Real-Time Performers architecture. This chapter focuses on algorithm aspects;
- Chapter 7 presents the exploitation of Real-Time Performers architecture and framework to the development of the system described in chapter 6;
- Chapter 8 draws some conclusions and presents future development about CLAM.

1.5 Acknowledgements

I wish to thank my mentor, advisor, and guide, Prof. Francesco Tisato, for his support and guidance throughout all the years from my degree in Computer Science till now. My fellow Ph.D. students, Sergio Ruocco, Andrea Trentini, and Valentina Sachero, that have supported me during all this years both from the scientific and the human point of view. Finally, I am very grateful to Dr. Fabio Marchese that introduced me in the mysterious and charming world of Robotics. Without him, part of this work would not be possible.

Chapter 2

State of the Art

2.1 Introduction

In this chapter we will provide an overview about localisation, world modelling, and both localisation and world modelling approaches devoted to support the autonomous navigation of a robot inside an unknown environment. Then, a survey of the state of the art about software architecture will be provided.

2.2 Robot Autonomous Navigation

Today robot are widely used in industry, in particular for tasks such as welding, painting, and packaging. All these robots are in form of manipulators that carry out repetitive movements.

There is currently a diffusion in the set of applications away from factory setting toward office and domestic applications. One issue that, in particular, motivates this change is the aging of the society.

Such kind of robot must include facilities for autonomous navigation in indoor environments. To navigate in autonomy a robot must be provided with methods helping it in self-localising inside the environment in which it operates. In other words, it must be able to respond to the question "Where am I?"

In industrial settings it is often permissible both to use artificial markers and to provide the robot with a map of the environment for localisation purpose. For operation in an office environment or in a domestic setting, it must not be acceptable to place artificial markers throughout the environment. In addition, the availability of a map is a difficult issue.

Consequently, there is a lot of interest in investigating methods allowing automatic acquisition of environments maps based on natural features with the purpose of exploiting them for robust localisation and navigation.

2.2.1 The Localisation Problem

For any task the robot might do it should be able to self-localise, i.e., it should be able to localise its place in the world.

Robot localisation, often referred to as position estimation or position control, is currently a highly active field of research since it has been recognised as one of the most fundamental problems in mobile robotics [20] and [10].

The problem of self-localisation is divided into two main categories: *global* and *local*[1]. In global localisation, the robot should be able to estimate the position without any *a priori* information about the position itself. In local localisation, the robot knows the starting position within some certainty and then tries to keep track of the position while moving. In detail:

- *Local Localisation (or Pose Tracking).* In many applications an initial estimate of the robot pose is known. During the execution of a task, a robot must update this estimate using measurements from sensors. Using only sensors measuring relative movements (e.g., odometry), the error in the pose estimate increases over time as errors are accumulated. Indeed, the information provided by these sensors can be integrated over time to give an estimate of the robot pose, that is valid when moving over short distance. This is not true for longer distances: the errors present in the measurements will accumulate and result in an unbounded pose estimation error. Therefore other sensors are needed to bound this error, and, so, to provide a more accurate information about the robot pose. This is achieved by matching the measurements provided by adjunctive sensors with an environmental model. Even if the matching problem (or the problem of the correspondences) is one of the hardest problem in any estimation process, in pose tracking is simplified since in general a good initial position is provided and, consequently, only a relatively small region around the robot pose may be considered;

- *Global Localisation.* In some situation, and for a really autonomous robot, the initial estimate position is not available. The process of finding the pose without no (or very limited) initial estimate of position, is called *global localisation* or *pose initialisation.* Techniques of this type solve the so-called wake-up robot problem, in that they can localise a robot without any prior knowledge about its position.

  They furthermore can handle the *kidnapped robot problem* [39], in which a robot is carried to an arbitrary location during it's operation. The wake-up problem is the special case of the kidnapped robot one in which the robot is told that it has been carried away.

[1]With local localisation being a part of the global one.

Obviously, the problem is more hard than the pose tracking one, specially with the data association process. The level of complexity increases with the size of the environment and its symmetry.

2.2.2 The Map Acquisition Problem

The problem of environment mapping may be sumarised as the problem of acquiring a spatial model of a robot environment when it is not provided. A robot needs a consistent map of the environment to self-localise (when using sensors measuring relative movements) and to plan motion. As the environment is unknown, robot should be able to construct the map in an incremental way. To acquire a map, robots must possess sensors perceiving the outside world. Sensors commonly used for this task include cameras, range finders using sonar, laser, and infrared technology, radar, tactile sensors, compasses, and GPS. However, all sensors are subject to errors, often referred to as measurement noise.

The problem of map acquisition (also referred as *map building*) presents completed issues, like:

- *The measurement noise.* Modelling problems, such as robotic mapping, are usually relatively easy to solve if the noise in different measurements is statistically independent. If this were the case, a robot could simply take more and more measurements to cancel out the effects of the noise. Unfortunately, in robotic mapping, the measurement errors are statistically dependent. This is because errors in control accumulate over time, and they affect the way future sensor measurements are interpreted. As a result, whatever a robot infers about its environment is plagued by systematic, correlated errors. Accommodating such systematic errors is the key to build maps successfully, but it is also a complicating factor in robotic mapping. Many existing mapping algorithms are therefore surprisingly complex, both from a mathematical and from an implementation point of view.

- *Entities high dimensionality.* The second complicating aspect of the robot mapping problem arises from the high dimensionality of the entities that are being mapped.

- *Matching problem.* This problem, also known as the data association problem, consists in determining if sensorial measurements taken at different points in time correspond to the same physical object in the world.

- *Environmental dynamics.* Environments change over time. The changes may be relatively slow (e.g., structural changes), or faster (e.g., door status changes).

2.2.2.1 Map Representation Models

There are many ways to represent the knowledge about an environment. The representation is strictly related to the typology of the sensors adopted for the map construction.

In the following, we briefly introduce two common classification criteria (for details see [75] and [72]):

- *A generic classification.*
  - *Topological map.* The environment is represented as a graph where nodes represent places of importance (e.g., room), meanwhile arcs correspond to connections among places (e.g., corridors);
  - *Features maps.* Geometric features are used to represents the environment. Feature examples are points and lines:
  - *Grid maps.* The environment is divided into a grid in which each cell represents a small area (or volume) of the environment. Every cell is characterised by a certainty value expressing its occupancy;
  - *Appearance maps.* Sensor data is used directly to form a function from sensor data to pose space.
- *A classification oriented to path-planning.*
  - *Paths Maps.* The executable robot paths are identified off line and then memorised in a graph or by means of sequences of motion commands;
  - *Free-space Maps.* This method consists in building a map by means of sensorial surveys of a robot that moves in a completely unknown environment. The paths are constituted by rectilinear segments separated by halt-points in which a direction change is made. These paths are memorised in a spatial graph, in which the links represent the executable trajectories, the nodes represent free areas or possible obstacles;
  - *Object Oriented Maps.* These maps are exploited specially in case of completely known environment, constituted by fixed obstacles, or having available a system able to effect a geometric reconstruction of the obstacles. In this case the map could be realised by means of a list of objects, each of which is described as a list of segments, that is of couples of coordinates (also in probabilistic terms) of the vertexes referred to the adopted coordinates;
  - *Composite Space Maps.* The methods described until now has the remarkable characteristic to use high level information (geometric/topologic) relative to the environment and to the objects

inside. In many applications it is instead necessary to take care of a lower level information due to less informative sensors. In such situations, this model is the right choice. It divides the world in grids of cells, containing an attribute that specifies the state (typically empty or free cell), also using probabilistic variables (Occupancy Grids).

2.2.3 Closing the Loop: the SLAM Problem

An autonomous robot must perform both pose estimating and map reconstruction. Since pose estimation requires a map and mapping requires the pose, there ia a "chicken and egg" problem [65]. The answer to the question is that they have to be carried out at the same time. The problem may be stated as follows: starting from an initial position, a mobile robot travels through a sequence of positions and obtains a set of sensor measurements at each position. The goal is for the mobile robot to process the sensor data to produce an estimate of its position while concurrently building a map of the environment. This means that both the map and the robot position are not known. In this case the robot start in an unknown location in an unknown environment and proceed to incrementally build a navigation map of the environment while simultaneously use this map to update its location.

In this problem, robot and map estimates are highly correlated and cannot be obtained independently of one another.

This problem is usually known as Simultaneous Localization and Map Building (SLAM) and was was first proposed by Leonard and Durrant-Whyte [65]. The problem presents a number of complicated issues, including:

1. efficient mapping of large-scale environments;
2. correct association of measurements;
3. robust estimation of map and vehicle trajectory information.

2.2.4 Techniques for Mobile Robot Localisation

Most of the techniques for mobile robot localization in the literature belong to the class of local approaches or tracking techniques, which are designed to compensate odometric error occurring during navigation. But, as cited by Borenstein [10], perhaps the most important result from surveying the literature on mobile robot positioning is that, to date, there is not truly elegant solution for the problem.

Most of the partial solutions can roughly be categorized into two groups: *relative* and *absolute* position measurements. Because of the lack of a single good method, developers of mobile robots usually combine two methods, one from each group. The two groups can be further divided into the other categories: odometry, inertial navigation, active beacons localisation, global positioning systems, landmark navigation, model matching.

Another way to categorise localisation techniques is to consider the approach to solution: there are method that consider the measurements uncertainty, other that do not take into account. The former can be divided in three subcategories again: techniques that use Kalman filtering, other that adopt Markov approach, and the last Monte Carlo method.

In the following a brief overview will be given.

2.2.4.1 Categorisation on the basis of the position measurements

- *Relative position measurements (also called dead-reckoning)*
  - *Odometry.* This method uses encoders to measure wheel rotation and/or steering orientation. As reported in [19], odometry presents both advantages and disadvantages. It is totally self-contained, and provide the vehicle with an estimate of its position. On the contrast, the position error grows without bound unless an independent reference is used periodically to reduce the error.
  - *Inertial navigation.* This method uses gyroscopes and accelerometers to measure rate of rotation and acceleration. Measurements are integrated once (or twice) to yield position. Inertial navigation systems, as odometry, are self-contained. On the downside, inertial sensor data drifts with time because of the need to integrate rate data to yield position; any small constant error increases without bound after integration. Inertial sensors are thus unsuitable for accurate positioning over an extended period of time. Another problem with inertial navigation is the high equipment cost.
- *Absolute position measurements (reference-based systems)*
  - *Active beacons localisation.* This method computes the robot absolute position from measuring the direction of incidence of three or more actively transmitted beacons. The transmitters, usually using light or radio frequencies, must be located at known sites in the environment.
  - *Artificial landmark recognition.* Similarly to active beacon, in these methods, artificial landmarks are placed at known locations in the environment. The advantage is that they can be designed for optimal detectability even under adverse environmental conditions, and that the position errors are bounded. The disadvantages are that they can not be used in all the environment, that

  three or more landmarks must be visible to allow position estimation, and that the detection of external landmarks and real-time position fixing may not always be possible.

  - *Natural landmark recognition.* This approach is based on the consideration that landmarks are distinctive features in the environment. These techniques present the advantages that the environment do not have to be modified, but the environment must be known in advance. The reliability of this method is not as high as with artificial landmarks.
  - *Model matching.* A robot hosts sensors to acquire information about the environment. The acquired information is compared to a *map* or *world model* of the environment. The robot absolute location can be estimated if features from the sensor-based map and the world model map match. Map-based positioning often includes improving global maps based on the new sensory observations in a dynamic environment and integrating local maps into the global map to cover previously unexplored areas. One of the most important and challenging aspects of map-based navigation is map matching, i.e., establishing the correspondence between a current local map and the stored global map.

2.2.4.2 Categorisation of the basis of the method

- *Techniques without uncertainty representation.* In work [120], angle histograms constructed out of laser range-finder scans and taken at different locations in the environment are stored. The position and orientation of the robot are calculated by maximizing the correlation between the stored histograms and laser range-scans obtained while the robot moves through the environment.
  The estimated position, together with the odometry information, is then used to predict the position of the robot and to select the histogram used for the next match. Other works like [123] and [98] present similar techniques as [120], but using hill-climbing to match local maps built from ultrasound sensors into a global occupancy grid map. The location of the robot is represented by the position yielding the best match.

- *Probabilistic approach.*

  - *Kalman filter techniques.* Kalman filters [59] are a signal processing technique widely used in robot systems. Many works adopting the Kalman filtering to position estimation are similar in the model used to represent the robot motion. They differ mostly

in the way they use sensorial inputs to update the gaussian distribution. Works like [74] and [107], the robot position is represented using a Gaussian distribution over the three-dimensional state-space of the robot. The mode of this distribution yields the current position of the robot, and the variance represents the robot uncertainty. Whenever the robot moves, the Gaussian is shifted according to the distance measured by the robot odometry. Simultaneously, the variance of the Gaussian is increased according to the model of the robot odometry. New sensory input is incorporated into the position estimation by matching the perceptions with the world model.
In [64], the beacons extracted from sonar scans are matched with beacons predicted from a geometric map of the environment. These beacons consist of planes, cylinders, and corners.
In [19], the author update the current position estimate, matching distances measured by infrared sensors with a line segment description of the environment.
In [97], many techniques for tracking purpose and based on occupancy grid maps and ultrasonic sensors are compared. They show that matching local occupancy grid maps with a global grid map results in a similar localization performance as if the matching is based on features that are extracted from both maps.
[99] compares the robustness of two different matching techniques with different sources of noise. The authors suggest the adoption of a combination of map-matching and feature-based techniques in order to inherit the benefits of both.
Both [50] and [69] use a scan-matching technique to estimate the position of the robot based on laser range-finder scans and learned models of the environment. [4] adopt a similar technique but providing an high accuracy estimate of robot position.
All these works assume that robot position can be represented by a single Gaussian distribution. The advantage of Kalman filter-based techniques lies in their efficiency and in the high accuracy that can be obtained. The restriction to a unimodal Gaussian distribution, however, is prone to fail if the position of a robot has to be estimated from scratch, i.e. without knowledge about the starting position of the robot.
Furthermore, these technique are typically unable to recover from localization failures. Recently, [56] introduced an approach based on multiple hypothesis tracking, which allows to model multimodal probability distributions as they occur during global localization.

– *Markov approaches*. There are many works that solve the localisation problem adopting Markov approach. We can cite [87], [103], [57], [12], [52], and [88]. The different variants of these technique can be roughly distinguished by the type of *discretisation* used for the *representation* of the state space: [87], [103], and [57] use Markov localization for landmark-based navigation, and the state space is organized according to the topological structure of the environment. Here nodes of the topological graph correspond to distinctive places in hallways such as openings or junctions and the connections between these places. Possible observations of the robot are, for example, hallway intersections. The advantage of these approaches is that they can represent ambiguous situations and thus are in principle able to globally localize a robot. Furthermore, the coarse discretisation of the environment results in relatively small state spaces that can be maintained efficiently. The topological representations have the disadvantage that they provide only coarse information about the robot's position and that they rely on the definition of abstract features that can be extracted from the sensor information. The approaches typically make strong assumptions about the nature of the environments. [87], [103], and [57] for example, only consider four possible headings for the robot position assuming that the corridors in the environment are orthogonal to each other.

2.2.5 Techniques for Map Matching

Robot mapping research has a long history (see [114]). In the first research years and till early 1990s, the main approaches to the problem were divided into two categories: *metric* and *topological* approaches. The former captures the geometrical aspects of the environment, the latter describe the environment as as a graph where nodes are the places and arcs connect them.

Examples of early metric approaches were proposed in [84], [37], and [38]. In the above cited works the authors proposed an algorithm (*occupancy grid mapping algorithm*) that represents the environment maps by grids modelling the free and the occupied spaces of the environment. This approach has been widely used in robot system such as [11], [51], and [124].

Another metric oriented algorithm based upon the use of polyhedra to describe the environment may be found in [17].

Concerning topological representation, there are several works. Some examples may be found in [62], [18], and many others.

Historically, another classification criteria about mapping algorithms was based upon the coordinate system used to express the position of the entities: *robot-centric* and *world-centric*. The former expresses entities in the reference frame the origin of which is the robot, the latter in an absolute reference frame.

From the 1990s years, the algorithms for map matching are based upon probabilistic approaches. These approaches may be divided into three major categories: approaches exploiting *Kalman filtering*, approaches exploiting *expectation maximization*, and approaches based upon *objects* identification ([114]).

Kalman filtering approaches are used to estimate the environmental maps by usually describing the location of landmarks or significant features (see, for example, [67] and [63]).

Expectation maximization approaches specifically address the correspondence problem in mapping, i.e., the problem of determining whether sensor measurements taken at different instant in time correspond to the same real entity in the environment (see, for example, [100] and [113]).

Finally, the object identification approaches are based upon the recognition of specific object in the environment such as doors, ceilings, and so on (see, for example, [68] and [73]).

In general and with camera sensors, matching is achieved by first extracting features, followed by determination of the correct correspondence between image and model features, usually by some form of constrained search [19].

2.2.5.1 Feature-based Visual map Building

We consider in the following the problem of constructing features map using camera sensors. The map update process may be decomposed in the following two problem:

- *features matching* performed on features (usually points or areas)present in both the image and the model frames.
- *model-to-image transformation* involving the translation, and rotation between the 3D model and the sensed data reference frame.

A very exhaustive survey of model-to-image registration is give in [31].

The problem of *features matching* have been thoroughly explored in the last years. Almost all the proposed approaches are based upon the ICP (Iterated Closest Point) [8]:
Let us suppose to have two sets of 3D points which correspond to a single shape but are expressed in different reference frames. We call one of these sets the *model set X*, and the other the *data set Y*.
They assume that *for each point in Y, the corresponding point in X is known.*
The problem is to find the 3D transformation which, when applied to the data set Y, minimizes a distance measure between the two point sets.
More formally, the goal is:

$$\min_{R,t} \sum_{i=1}^{N} \|x_i - (\mathbf{R}y_i + t)\|^2 \tag{2.1}$$

where **R** is a 3×3 rotation matrix, t is a 3×1 translation vector, and the subscript i refers to the corresponding elements of the sets X and Y.
In general, however, point correspondence are unknown: for each point y_i from the set Y, there exists at least one point on the surface of X which is closer to y_i than all other points in X. This is the *closest point*, x_i. The basic idea behind the ICP algorithm is that, under certain conditions, the point correspondence provided by sets of closest points is a reasonable approximation to the true point correspondence.
The author of [8] proved that if the process of finding closest point sets and then solving Eq. (2.1) is repeated, the solution is guaranteed to converge to a local minimum. The ICP algorithm can be summarised as follows:

1. For each point in Y, compute the closest point in X;
2. With the correspondence from step 1, compute the incremental transformation (**R**, t) with SVD;
3. Apply the incremental transformation from step 2 to the data Y;
4. If the change in *total mean square error* is less than a threshold, terminate. Else goto step 1.

The algorithm based upon ICP differ in terms of:

- *Considered features.*
  - Point to point registration. Data are disaggregated ([8] and [85])
  - Volumetric scene ([30] and [34])
  - Segment to segment registration([60])
- *The point correspondence.*
  - Minimum Euclidean point-to-point distance ([8], and [125])
  - Minimum point-to-tangent plane distance ([121])
  - Weighted minimum point-to-plane distance ([29])
  - Inverse camera calibration ([9])
- *The detection and handling of the outliers.* Some points correspondences come out wrong due to noise, occlusions, and misalignment in the current estimate. Consequently, the detection and handling of such outliers is important.
  - Threasholding based on point distance statistic computed between [125] and within [28] views
  - Point-pair compatibilities computed from colour ([47])
  - covariance-matrix-based affinities ([119])

- *The mathematical description of the transform components.* Once point correspondences have been established, the rigid transformation have to be computed. The methods are based upon a number of least-square method (all in closed form). For a comparison see [36].
  - Singular value decomposition of a matrix (SVD) ([5])
  - Orthonormal matrices ([55])
  - Unit (or dual) quaternions ([54] and [118])
- *Pose determination target.*
  - Object in a scene
  - The overall scene
- *Sensor typology.*
  - Acoustic
  - Camera
  - Sonar
  - Range-finder
  - Laser
  - ...

2.2.5.2 Updating algorithms

The typology of information, the sensor characteristics, the map representation, and the presence of data association solution, are the main differences between them.

Most of the proposed mapping algorithms are probabilistic. Some algorithms are incremental, and hence can be run in real time, whereas others require multiple passes through the data.

Some algorithms require exact pose information to build a map, whereas others can do so using odometry measurements.

Some algorithms are equipped to handle correspondence problems between data recorded at different points in time, whereas others require features to carry signatures that makes them uniquely identifiable.

The reason for the popularity of probabilistic techniques arises from the fact that robot mapping is characterized by uncertainty and sensor noise. Probabilistic algorithms approach the problem by explicitly modeling different sources of noise and their effects on the measurements. In the evolution of mapping algorithms, probabilistic algorithms have emerged as the sole winner for this a so difficult problem.

Some of the approaches proposed are listed in the following:

- *Kalman Filter Approaches.* A classical approach to generating maps is based on Kalman filters [59]. This approach can be traced back to a highly influential series of papers by Smith, Self, and Cheeseman [107] and [108]. The authors proposed a mathematical formulation of the approach that is still in widespread use today.

- *Expectation Maximization Algorithms.* A recent alternative to the Kalman filter paradigm is the *expectation maximization* family of algorithms (EM). EM is a statistical algorithm that was developed in the context of maximum likelihood estimation with latent variables, in a seminal paper by Dempster, Laird and Rubin [21]. In [115] and [13] is reported that EM algorithms constitute the best solutions to the correspondence problem in mapping. In particular, EM algorithms have been found to generate consistent maps in non optimal conditions. The main disadvantage is that EM algorithms do not retain a full notion of uncertainty: they look in the space of all maps, in an attempt to find the most likely map. To do so, they have to process the data multiple times. Hence, EM algorithms cannot generate maps incrementally, as is the case for many Kalman filter approaches.

In general, the presented approach address the mapping problem with unknown robot pose. Other approach, like Occupancy grid Maps (previously proposed in [84] and [38]) and Object Maps, deal with the problem of mapping with *known* poses.

2.2.6 Techniques for SLAM problem resolution

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in mobile robotics: while a robot navigates in an unknown environment, it must incrementally build a map of its surroundings and localize itself within that map [114].

During the past few years significant progress has been made towards the solution of the SLAM problem.

Initial work by Smith [106] and Durrant-Whyte [33] established a statistical basis for describing geometric uncertainty and relationships between features or landmarks. At the same time Ayache and Faugeras [7], and Chatila and Laumond [17] were undertaking early work in visual navigation of mobile robots using Kalman filter-type algorithms.

Discussions on how to solve the SLAM problem have their response soon after in the key paper by Smith, Self and Cheeseman [107]. This paper showed that as a mobile robot moves through an unknown environment taking relative observations of landmarks, the estimates of these landmarks are all necessarily correlated with each other because of the common error in estimated vehicle location.

Work then focused on Kalman-filter based approaches to indoor vehicle navigation. Especially: Leonard and Durrant-Whyte with work on sonar and data association [66], and Faugeras with work on visual navigation/motion [40].

In 1991 "Chicken and Egg" paper [65] identified some of the key issues in solving the SLAM problem. A realisation that the two problems must be solved together is dated around 1991.

The SLAM acronym coined in 1995 during the ISRR (International Symposium of Robotics Research) conference.

The first proofs of convergence and the first demonstrations of the SLAM algorithm are presented by Leonard and Feder in their work with sonar [42]. Dissanayake, Newman, and other authors in their work on outdoor radar and sub-sea [27] provide the final convergence proofs.

The first session on navigation and SLAM problem was held in the ISRR conference in 1999. It was a key event. Some of subsequent works are on large-scale implementations [32], data association [14], understanding the applicability of probabilistic methods [111] and [113], multiple vehicle SLAM [122]), implementations indoor, on land, air and sub-sea.

Most of the approaches treat SLAM as a Kalman filtering problem. The system state at time t is given by 2.2, where x_t is the robot pose at time t, l_i is the state of landmark i (landmarks are assumed stationary) and n_t is the number of landmarks observed up to time t.1 The length of the state vector will increase over time, as more landmarks are observed. The filtered belief state at time t, $p(\mathbf{m}_t|observationstotimet)$, is a multivariate Gaussian distribution $N(\mu_t, \Sigma_t)$. Its mean μ_t can be viewed as an estimate of the map $\mathbf{m}_t$, and its covariance matrix Σ_t as a measure of its estimation confidence.

$$\mathbf{m}_t = \begin{bmatrix} \mathbf{x}_t \\ \mathbf{l}_1 \\ . \\ . \\ . \\ \mathbf{l}_{nt} \end{bmatrix} \tag{2.2}$$

The Kalman filter solution is elegant, but it does not scale to large SLAM problems. Indeed, the Kalman filter approach comes with a number of limitations. Most notably, the inability to represent complex environment or feature models, the difficulty of faithfully describing highly skewed or multi-modal vehicle error models, and the inherent complexity of the resulting data association problem.

A parallel approach to vehicle navigation, which overcomes many of these limitations, is to consider navigation as a Bayesian estimation problem [111]. In this method, vehicle motion and feature observation are described directly in terms of the underlying probability density functions and Bayes theorem is used to fuse observation and motion information. Practically, these methods are implemented using a combination of grid-based environment modelling and particle filtering techniques. These Bayesian methods have demonstrated considerable success in some challenging environments [112].

2.3 Software Architectures

Software Architecture is a discipline within the Software Engineering field emerged in the middle 80s, and that has been raising increasing interest ever since. Mary Shaw and David Garland may be considered the pioneers of that discipline writing works like [46], [1], [45], and [101] that have significantly contributed to the widespread attention paid to the discipline. Their work resulted in the book [102] that, even if dated 1996, it may be considered the most authoritative (and elsewhere cited) reference concerning the software architecture.

Software architecture discipline proposes a new set of concepts for describing, and reasoning about, software compositions at the high level of abstractions at which software architects conceive software systems. In other words, software architecture should represent an high-level view of the system revealing the structure, but hiding all implementation details. Specifically, it should publish attributes such as responsibilities (of the constituents of the architecture), distribution, and deployment.

Abstractly, software architecture involves the description of: elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. More in detail, an architectural description of a software system should identify:

- the partition of the the overall functionality into *component*s;
- the behaviour of components;
- the protocols used by components to communicate and cooperate i.e., which *connector*s exist between them.
- the overall *topology* of the system, i.e., which components and connectors the system is made up of;
- the overall *strategy* of the system, i.e., temporal and functional dependencies among activities carried out by different components.

Finding an appropriate architectural design for a system is a key element for its long-term success. Architectural descriptions serve as a skeleton around which system properties may be modelled, and thereby serve a vital role in exposing the ability of system to meet its gross system requirements. The adoption of an architecture leads to clarify intentions, makes decisions and implications explicit, and allows system-level analysis. All these benefits are reflected in the Maintenance phase of the software life-cycle decreasing its costs.

The big innovation concerning software architecture is that it allows software engineer focusing on the overall structure of software systems instead of on individual data structures and algorithms. The components described by architectural descriptions represent high-level abstractions useful either for the application domain or for structuring the system itself. Due to the fact that at this level of thinking the system both domain-related concepts and high-level structuring concepts are employed, it is often mentioned in the literature that this level of design conceptually represents an intermediate step between the specification of requirements and the proper design [91].

What is still missing is a conceptual framework for expressing architectural concepts since it is too high-level to be represented by programming language concepts. A closely related problem is that of expressing architectural styles. The use of styles and patterns is one of the hallmarks of mature engineering disciplines. They are well-known design solutions to recurring problems. A classification of architectural style may be found in [102], while another concerning design pattern in [44].

2.3.1 Software Architecture Activities

The research activities related to Software Architecture may be placed into five major and interrelated areas.

2.3.1.1 Architectural Description Language (ADL)

This research area addresses the problem of architectural characterisation by providing new architectural description language. These languages are aimed at giving practitioners better ways of writing down architectures so that they can be communicated to others and in many cases analysed with tools. A complete survey of ADL may be found in [76].

2.3.1.2 Codification of current knowledge

This research area addresses codification of architectural expertise. Work in this area concerns with cataloguing and rationalising the variety of architectural principles and patterns that engineers have developed through software practice.

Every style describes a system architecture as a collection of components together with a description of the interactions among these (the connectors)[102]. More specifically, an architectural style defines:

- a *vocabulary* of component and connector types;
- a set of *constraints* on how they can be combined (topology rules);

Category	**Style**
Dataflow systems	Batch sequential
	Pipes and filters
	Process-control
Virtual Machines	Interpreters
	Rule-based systems
Call-and-return statements	Main program and subroutine
	O-O systems
	Layered
	Client-server
	Remote procedure calls
Data-centered systems (repositories)	Databases
	Hypertext systems
	Blackboards
Independent components	Communicating processes
	Event systems

Table 2.1: A taxonomy of architectural style

- one or more *semantic models* specifying how to determinate a system overall properties from the properties of its parts (not for all styles).

A taxonomy of architectural style is given in table 2.1.

2.3.1.3 Frameworks for specific domains

This research area of software architecture addresses architectural framework for a specific class of software. When successful, such frameworks can be easily instantiated to produce new products in the domain. In the most restricted definition, a framework is a set of modules that can be reused for the development of systems and embodying a certain architectural organisation, so that implementation of architectural concepts needs not be carried on from scratch each time a certain style is reused.

More interesting are the Object-Oriented frameworks: collections of classes implementing useful concepts that can be reused in a large class of applications. These classes constitute "*a generic software system for an application domain*" [92].

2.3.1.4 Formal models

This research area of software architecture addresses formal underpinnings for architecture. As new notations are developed, and as the practice of architectural design is better understood, formalisms for reasoning about architectural design become relevant.

2.3.1.5 Architecture recovery

More recent with respect of the previous areas, this research line addresses the problem of reverse engineering at the architectural level. The overall aim is that of constructing an architectural description of an existing system via analysis of the source code.

2.4 Reflection

A lot of powerful notations and models for describing architecture have been developed. These are used to specify architectures that will be implemented with traditional programming language. Since programming language concepts are at a lower abstraction level than architectural descriptions, it is often the case that architectural information is distributed, implicit, and intermixed with computational issues at code level [16].

To address this problem, in [16] is proposed a design model termed Architectural Programming-in-the-Large (APIL), whereby architectural abstractions are encapsulated in dedicated run-time entities. Computation is charged to architecture-unaware components whose activity is coordinated by architectural entities enforcing a specific architectural organisation onto the system.

In [15] the author proposes a novel approach to the dynamic inspection and modification of architectures, which formalises the ideas from APIL. The approach is based on *reflective* mechanism and is termed *Architectural Reflection*. The basic idea is that an *architectural reflective system* is aware of its own architecture, whereas its components are not aware of the system architecture. Architectural reflection is based upon previous works concerning *Computational Reflection* [70].

In the following more exhaustive definitions of both computational and architectural reflection are provided.

2.4.1 Computational reflection

Programming languages paradigms, design patterns, software architectures, middleware, and frameworks provide effective *abstractions* for specific aims. However, abstraction is defined as the process of eliminating details not relevant for the task at hand. When abstractions are applied to a computational system, details (better said, aspects) not relevant for the task at hand are either neglected or hidden inside inner system layers, which exploit best-effort strategies. But if the aim changes, previously hidden “details” may become relevant and claim for proper abstractions. Time and architecture cannot be neglected when designing a time-sensitive system. Applications with *time sensitive architectures* must explicitly deal with non-functional requirements. Therefore they need *abstractions for properly capture temporal aspects of computation.* Such abstractions should allow both the internal architecture and the temporal behaviour of the system to be observed and controlled at the application programming level. Of course, the abstractions should preserve a proper separation of concerns.

Computational reflection is “the activity performed by a computational system when doing computation about (and by that possibily affecting) its own state and computations” [70]. Computational reflection was originally introduced by Smith (see [104] and [105]). Smith defines a reflective system as “A computer system able to reason about itself” and “. . . ‘reflection’ in its most general sense [. . .] the ability of an agent to reason not only introspectively, about its self and internal thought processes, but also externally, about its behaviour and situation in the world” [104]. In other words, reflection allows a computer to deal in a systematical way with the diverse aspects of its own computations.

In order to reflect, a system “embeds a theory of the system in the system [. . . that. . .] beyond being descriptive and true [. . .] must be [. . .] causally-connected, so that accounts of objects and events are tied directly to those objects and events” [104]. In practice real reflective computer systems express such theory in terms of data structures representing aspects of the system to the system itself. Then it is possible to differentiate between ordinary computations dealing with data structures representing the application domain, and reflective computations dealing with data structures representing aspects of the system itself. The causal connection ensures that changes in the system self-representations by ongoing computations are reflected in the system actual state or behaviour, and changes by ongoing computations to the system state or behaviour may affect data structures.

Rather than with full computational reflection, however, most research focus on “[. . .] a limited and rather introspective notion of ‘procedural reflection’: self-referential behaviour in procedural languages” [104]. Note that “procedural” does not refer to the ordinary classification of programming languages ([105], p. 41), in fact, Smith made a reflective Lisp, and [70] “discusses an OOL with an architecture for procedural reflection”. Actually, “In a procedurally reflective programming language programs are executed not through the agency of a primitive and inaccessible interpreter, but rather by the explicit running of a program that represents that interpreter” [24]. Such a program is the “language processor”, and reifies its state in meta-entities that depend on the language being made reflective: environment and continuations [24], meta-objects [70], and so on; similarly, depending on the language, the causal connection is mantained by reflective processors, meta-objects, MOPs [61] and so on.

Since, strictly speaking, programming languages abstract from implementations of the physical processor, the subject of procedural reflection is not the computer system as a whole anymore, but rather the abstract evolution of computational processes in themselves as defined in and by the programming language. Procedural reflection is implicitly constrained in the domain of programming languages. It cannot properly cross the implementation boundary between the reflective processor of the language, which is a program, and the real processor, which is a physical component of the computer that executes it. This is also the case of some reflective systems that provide reflective access to the language processor. In fact, the meta-objects reifying the processor are, in turn, defined in terms of another procedural programming language [109].

Systems with time sensitive architectures deal instead with temporal aspects of computations, which depend on the physical properties of the underlying computational system as a whole. Therefore, in order to reflect on temporal aspects and other physical aspects of computations, systems have to be modelled by suitable meta-entities that properly represent them; in addition, causal connection requires appropriate mechanisms that keep these entities and the corresponding real properties of the system properly aligned.

Having a different purpose from procedural reflection, both the meta-entities and the mechanisms for causal connection will likely be different. To clearly mark these distinctions and to avoid confusion with terms from the restricted field of procedural reflection, from now on, we will term *architectural abstractions* those meta-entities representing aspects of the system beyond the reach of programming languages. Nonetheless, since architectural abstractions represent in a causally connected way aspects of the system to the system itself, computations over architectural abstractions are in effect *reflective* computations [116].

2.4.2 Architectural reflection

Architectural reflection is defined as the computation performed by a system about its own software architecture [15], [116]. An architectural reflective system is structured into two layers called architectural layers: an *architectural base-layer* and an *architectural meta-layer*. The base-layer is the "ordinary system, which is assumed to be costuited by:

- a set of *computational components*. These are architecture-independent i.e., they embed no architectural assumption;
- a set of architectural components embodying architectural behaviour, which is to be attached to computational components. These components shield the computational components so that the latter can actually be architecture-independent;
- a set of connectors ruling over the interaction among components;

- a program-in-the-large prescribing how components should be assembled and interact i.e., the architecture of the system, addressing both topology and strategy. The program-in-the-large [16] is executed at run-time by a dedicated virtual machine that rules over the instantiation and behaviour of components.

The architectural meta-layer maintains causally connected data structures reifying the architecture of the base-layer.

According to the concept of domain as used in [70], the domain of the base-layer is the systems application domain, while the domain of the architectural meta-layer is the software architecture of the base-layer.

Chapter 3

Concurrent Localisation And Mapping

3.1 Introduction

Concurrent Localisation And Mapping (CLAM in the sequel) is the approach we propose to solve the *chicken and egg problem* raised previously. The problem can be summarised as follow: to precisely locate itself, a mobile robot relies on a precise model of its environment; however, to build a precise environment model, the mobile robot relies on a precise knowledge of its location.

The CLAM approach treats the *Localisation* and *Modelling*[1] activities as *concurrent*, i.e., mostly independent and with different timing constraints. Sometimes the two activities must synchronise. If synchronisation is needed, the relative rates of the two activities can be adjusted in order to meet as soon as possible the synchronisation requirements.

The CLAM approach assumes that:

- the robot is equipped with device(s) providing an error-prone estimate of its position;
- the robot is equipped with device(s) providing perceptions about the environment;
- the environment the robot explores is modelled as a collection of objects whose position is defined in a geometrical reference system;
- the robot speed and direction can be controlled;
- the overall environment does not change[2].

[1]In the sequel we will use the term *Modelling* to refer the activity of building a representation of the environment.

[2]This assumption aims at simplifying the discussion. It could be relaxed by enriching the World Model with some dynamics.

Under the above assumptions, the key concepts of the CLAM approach are the following:

1. *Localisation* and *Modelling*, both relying on *Perception*, are the basic activities performed by a robot when exploring an unknown environment;

2. *Localisation* and *Modelling* operate on *separate information* and are subject to *different timing constraints*. Therefore they can be performed *concurrently* and with *independent timings*;

3. Localisation relies on information which loosely depends on the information generated by *Modelling*, and vice-versa. Therefore *Localisation* and *Modelling* must *synchronise* whenever a *criticality* arises, i.e., whenever the information an activity relies on is not reliable;

4. Synchronisation is controlled by a *strategy* which relies on the observation of the criticalities and drives the relative rates of the activities.

The basic conjecture of this thesis is that a proper separation of concerns should help breaking the chicken-and-egg loop. Of course, *Localisation* and *Modelling* are related. However, they act on different time scales, so that they can be considered as mostly independent activities which sometimes synchronise under the control of a suitable strategy. Informally speaking, the activities of a chicken-farm and of a fast-food selling fried chicken are obviously related. However, they can be independently managed, provided that their relative rates are kept under control on the medium-long term and that criticalities, if any (i.e., lack of chickens to be fried), are properly managed.

It is worth noting that the goal of this thesis is not to formally demonstrate either that the proposed approach leads to an optimal solution, or that it possibly leads to a solution at all. This is an exploratory work towards the construction of complex systems whose overall behavior emerges from the individual behavior of simple components, which loosely interact thanks to their different execution rates. The rest of the chapter details the above concepts.

In the following we use the Unified Modelling Language (UML) (for details the reader can refer to [43], or more exhaustively [93]), to formalise some of the key concepts presented in this chapter. This choice is motivated by the fact that UML diagrams have the advantage of being both high-level enough to be understandable and low-level enough for an experienced reader to grasp all the necessary details.

3.2 CLAM Activities

Localisation and *Modelling* are the basic CLAM activities:

- *Localisation* aim is to make the robot able to self-localise inside an environment which is assumed to be known by means of a (partial) model;
- *Modelling* aim is the construction of an environment representation under the assumption that the robot knows its position inside the environment.

Both activities exploit information provided by *Perception. Perception* activity gets information from all the sensorial devices, performs basic processing, and presents to both *Localisation* and *Modelling* a suitable representation of what the robot perceives. All the two activities exploit the same kind (in Object-Oriented parlance, the same classes) of information, i.e.:

- robot *pose*, i.e, its position and orientation in a reference frame;
- *view*, i.e., a set of *geometrically located objects*. Geometrically located objects[3] are characterised by their position in a reference frame.

A robot *pose* is defined by a translation on the (x, y) plane and a rotation around the z axis (see equation 3.1).

$$P = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} \tag{3.1}$$

A *view* is defined as a set of geometrically located objects characterised by their position in a reference frame (see equation 3.2).

$$V = \{glo_i\} \quad | \quad i \in \mathbf{N} \tag{3.2}$$

Given a robot *pose* P and a *view* V, we define a *located view* as the pair in equation 3.3.

$$LV = (V, P) \tag{3.3}$$

A *located view* LV relates the objects in V to the robot *pose* P in which they have been, or could be, perceived[4].

A robot *pose* provided by specific devices (e.g, the odometric system) is called *perceived pose*. Its definition is given in equation 3.4

$$PP = \begin{bmatrix} \Delta x_P \\ \Delta y_P \\ \Delta \theta_P \end{bmatrix} \tag{3.4}$$

[3]In the following the objects are assumed to be Segments, characterised by the coordinates of a couple of points. Different kinds of objects could be exploited, ranging from points through high-level domain entities.

[4]Objects in V and *pose* P should be referred to the same reference frame. The presence of multiple reference systems introduces the need for coordinates transformations. This issue will be discussed in detail in chapter 6, and is skipped here for clarity.

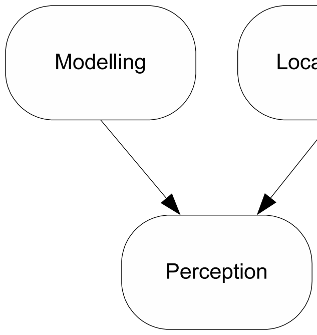


Figure 3.1: *Localisation*, *Modelling*, and their dependency on *Perception*

A set of geometrically located objects captured by specific devices (e.g., cameras, sonar, and so on) when the robot *pose* was defined by PP is called *perceived view*. Its definition is given in equation 3.5.

$$PV = \{pglo_i\} \quad | \quad i \in \mathbf{N} \tag{3.5}$$

Given the equations 3.4 and 3.5, we define as *perceived located view* the pair in equation 3.6.

$$PLV = (PV, PP) \tag{3.6}$$

Perception activity generates *perceived located view*s, i.e., sets of PLVs

Figure 3.1 emphasises the dependency of both *Localisation* and *Modelling* on the *Perception* activity. Note that, though both *Localisation* and *Modelling* rely on the same kind (i.e., classes) of information generated by *Perception*, they exploit different sets (i.e., instances) of information inside different timing frames. This is a basic remark to face the chicken-and-egg problem, as we shall discuss later.

3.2.1 Localisation

The localisation problem arises from the fact that the robot *perceived pose* PP, as generated by the odometric system, is prone to error. The *Localisation* activity aims at correcting the errors.

We define *reference pose RP* as a robot pose which is assumed to be correct. Its definition is given in equation 3.7.

$$RP = \begin{bmatrix} \Delta x_R \\ \Delta y_R \\ \Delta \theta_R \end{bmatrix} \tag{3.7}$$

We define also *reference view RV* as a set of geometrically located objects as perceived by the robot with pose RP (see equation 3.8).

$$RV = \{rglo_i\} \quad | \quad i \in \mathbf{N} \tag{3.8}$$

A *reference located view* is, then, the pair given in equation 3.9.

$$RLV = (RV, RP) \tag{3.9}$$

The *Localisation* activity corrects the odometric errors by exploiting the following information:

- the *current perceived located view* PLV;
- a *reference located view* RLV.

Defining *estimated robot pose* as the robot pose after the odometric error has been corrected (see equation 3.10), then *Localisation* also generates *estimated located views* ELV, i.e., the pair (PV, EP).

$$EP = \begin{bmatrix} \Delta x_E \\ \Delta y_E \\ \Delta \theta_E \end{bmatrix} \tag{3.10}$$

The idea behind the *Localisation* activity is quite simple. Assume that the *reference located view* RLV is correct, i.e., the robot with pose RP did observe the objects belonging to RV.

Let P_{RT} be the perceived rototranslation which leads the robot from the *reference pose* RP to the *perceived pose* PP. If PP has been properly perceived, by applying P_{RT} to the geometrically located objects in PV should produce a set $PV' = P_{RT} \cdot PV$ such that $PV' = RV$, i.e., each object in PV' is similar[5] (i.e., it exhibits similar geometrical features) to the corresponding object in RV.

If this does not hold, a rototranslation D_{RT} must be identified, such that $D_{RT} \cdot PV' = RV$. D_{RT} is the perception error of the odometric system. Therefore the proper *current located view* can be estimated as $ELV = (PV, EP)$ where $EP = D_{RT} \cdot PP$ is the *estimated pose* computed by the *Localisation* activity.

The obvious consequence is that EP is exploited to tune up the odometric system, and ELV is exploited as the *reference view* RLV for the next localisation step. Of course, there are several problems, some of them are local to the *Localisation* activity:

1. *Normalisation*: to relate PLV and RLV to a common reference frame;
2. *Association*: to identify, for each object in PV, a corresponding object in RV[6];
3. *Registration*: to estimate D_{RT} from pairs of corresponding objects in PV and RV and to generate ELV.

[5]The concept of "similarity" will be discussed in detail in the following.

[6]As a matter of fact, Association will be considered to be successful if a reasonable subset of the objects in RV and PV can be associated.

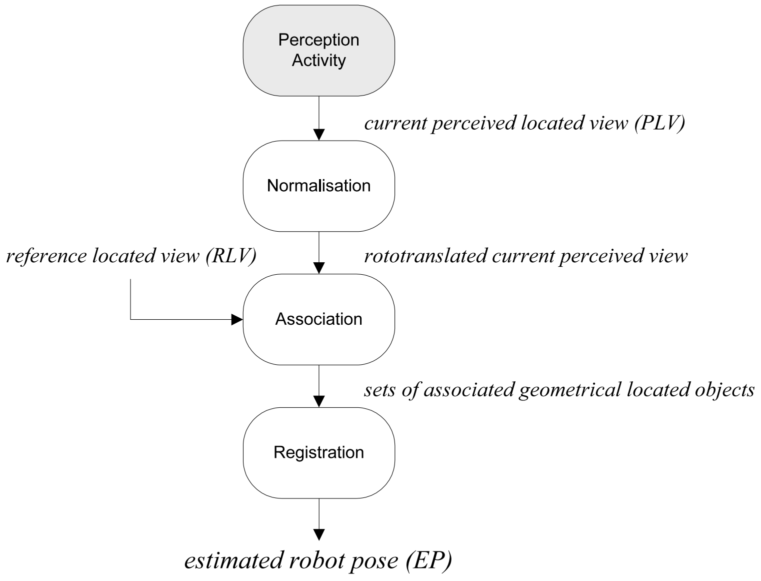

Perception Activity
current perceived located view (PLV)
Normalisation
reference located view (RLV)
rototranslated current perceived view
Association
sets of associated geometrical located objects
Registration
estimated robot pose (EP)

Figure 3.2: The *Localisation activity*

Solutions to these local problems turn into steps of the *Localisation* activity (see figure 3.2) and will be discussed in chapter 6.

The key issue is that if they are performed successfully, the *Localisation* activity can autonomously continue its task. A global problem arises whenever *Localisation* fails. This means that the *perceived located view PLV* cannot be successfully matched against the *reference located view RLV*. Apart from *Perception* failures (hardware faults, blackouts, image processing uncertainties, and so on) which are out of the scope of this work, *Localisation* fails if the *reference located view RLV* is not a plausible representation of the environment at the moment the *perceived located view PLV* was captured (e.g. the robot turn around a corner, or entered an unexplored part of the environment). This is what we call a *criticality*, which implies (at least) a synchronisation between *Localisation* and *Modelling*, as we shall discuss in section 3.4.

3.2.2 Modelling

Modelling aim is to incrementally build the *world model*, under the assumption that *Localisation* properly works.

The *world model* is defined as a collection of geometrically located objects, which rose to the rank of being representative of "real world" objects, whatever they are (see equation 3.11).

$$WM = \{wmglo_i\} \quad | \quad i \in \mathbf{N} \tag{3.11}$$

We define with the term *views history* a collection of *estimated located view* which are assumed to be (nearly) correct (see equation 3.12).

$$VH = \{ELV_i\} \quad | \quad i \in \mathbf{N} \tag{3.12}$$

Modelling exploits a *views history* VH and incrementally updates a *world model* WM.

Again, the underlying idea is quite simple. If there is a set of views ELV belonging to VH, and a reasonably[7] large set of objects glo_i such that $glo_i \in EV_i$ (with $EV_i \in ELV_i$) and all the objects exhibit similar geometric features, then a *wmglo* object can be added (if it does not already exist) to the *world model*.

As for *Localisation*, there are several local problems:

1. *Association*: to identify sets of corresponding objects glo_i in different views $ELV_i \in VH$[8];

2. *Fusion*: to merge the features of a set of corresponding objects in order to define the features of a *wmglo* object which is assumed to model a "real world" object with the purpose to update the *world model* WM;

3. *Integration*: to check whether the *wmglo* object already exists in WM, and to possibly increase the WM.

Solutions to these local problems turn into the steps of the *Modelling* activity sketched in figure 3.3, and will be discussed in the following. If they are performed successfully, the *Modelling* activity can autonomously continue its task.

A global problem arises whenever *Modelling* fails. This happens whenever sets of corresponding *glo* objects cannot be identified in the recent history, i.e. the view history VH is not a plausible representation of a suitable *world model*. Again, this is a *criticality* which implies a synchronisation between *Modelling* and *Localisation*.

Finally, the diagram in figure 3.4 sketches the CLAM classes of information (and their relationships).

`LocatedView` class is a composition of one `Pose` and one `View`. `Pose` and `View` may belong to *only* one `LocatedView` (filled diamond in diagram). This is the milestone of CLAM. This UML syntax specifies that instances created from both the `View` and the `Pose` classes may belong to only one instance of `LocatedView` class.

The same hold for both `View` and `WM`: they are compositions of one or more `GLO`s: the instance(s) composing `wm` (instance of `WM` class) and the

[7]What "reasonably large" does mean will be discussed in the following.

[8]This sub-activity may borrow from the results of the association sub-activity performed by *Localisation*.

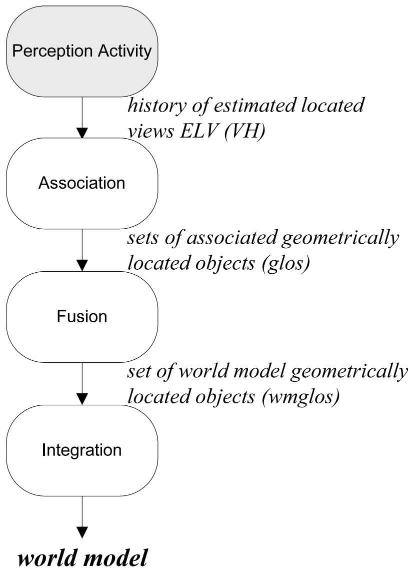

Perception Activity
history of estimated located views ELV (VH)
Association
sets of associated geometrically located objects (glos)
Fusion
set of world model geometrically located objects (wmglos)
Integration
world model

Figure 3.3: The *Modelling activity*

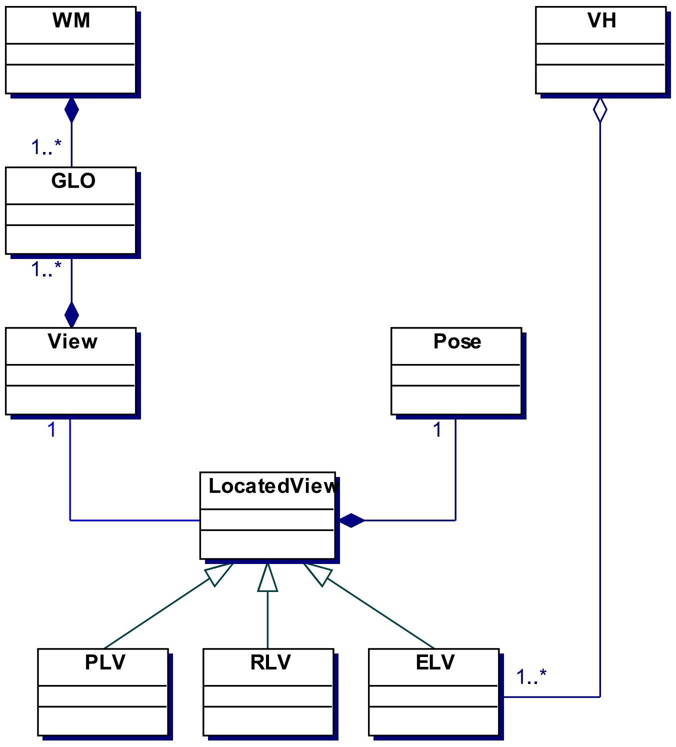


Figure 3.4: The classes of information in CLAM

instance(s) composing `view` (instance of `View` class) may only belong to `wm` and `view` respectively.

Finally, a view history `HV` exists if and only if (simple diamond in diagram) exists one or more `ELV`(s).

3.3 Concurrence

When criticalities do not arise during the execution of both *Localisation* and *Modelling*, the activities may be executed concurrently. As explained in section 3.2, for achieving the concurrence, the following conditions must hold:

- the *perceived located view* PLV can be successfully matched against the *reference located view* RLV;
- the ELV views belonging to the history VH must contain sets of corresponding *glo* objects.

When the two above conditions simultaneity hold, the activities do not operate on the same concrete sets (i.e., instances) of information and, consequently, may be executed concurrently. This is due to the fact that the information needed by one activity is not accessed by the other, i.e., no information is updated by more then one activity. In such situation a trivial pipeline between the activities may be exploited for synchronisation purpose.

Figure 3.5 sketches the UML activity diagram emphasising the concurrence of the CLAM activities. In the diagram, *swimlanes*[9] represents the single activities.

The execution of *Perception* activity produces a *perceived located view* PLV that is exploited by the *Localisation* activity. This is expressed by the dashed line[10] connecting the *perceived located view* (pv:PLV in the diagram) to the Normalisation sub-activity[11].

Association_L sub-activity reifies in the diagram, the Association step in the *Localisation* activity. The arrow connecting Normalisation and Association_L sub-activities is called *transition* and specifies the execution sequence. In this specific case, the Association_L sub-activity follows the Normalisation one. Association_L exploits the *reference located view* (the box rv:RLV in the diagram). This is represented by the dashed line connecting rv:RLV to the sub-activity.

[9]In UML activity diagram, a *swimlane* models a single activity of the system.

[10]A dashed line connecting objects (the boxes) to activity (the ovals) specifies the object flow.

[11]Even if Normalisation, Association, and so on, are, in UML parlance, activity, we refer them with the term sub-activity to not bring confusion with *Perception*, *Localisation*, and *Modelling* (called activities).

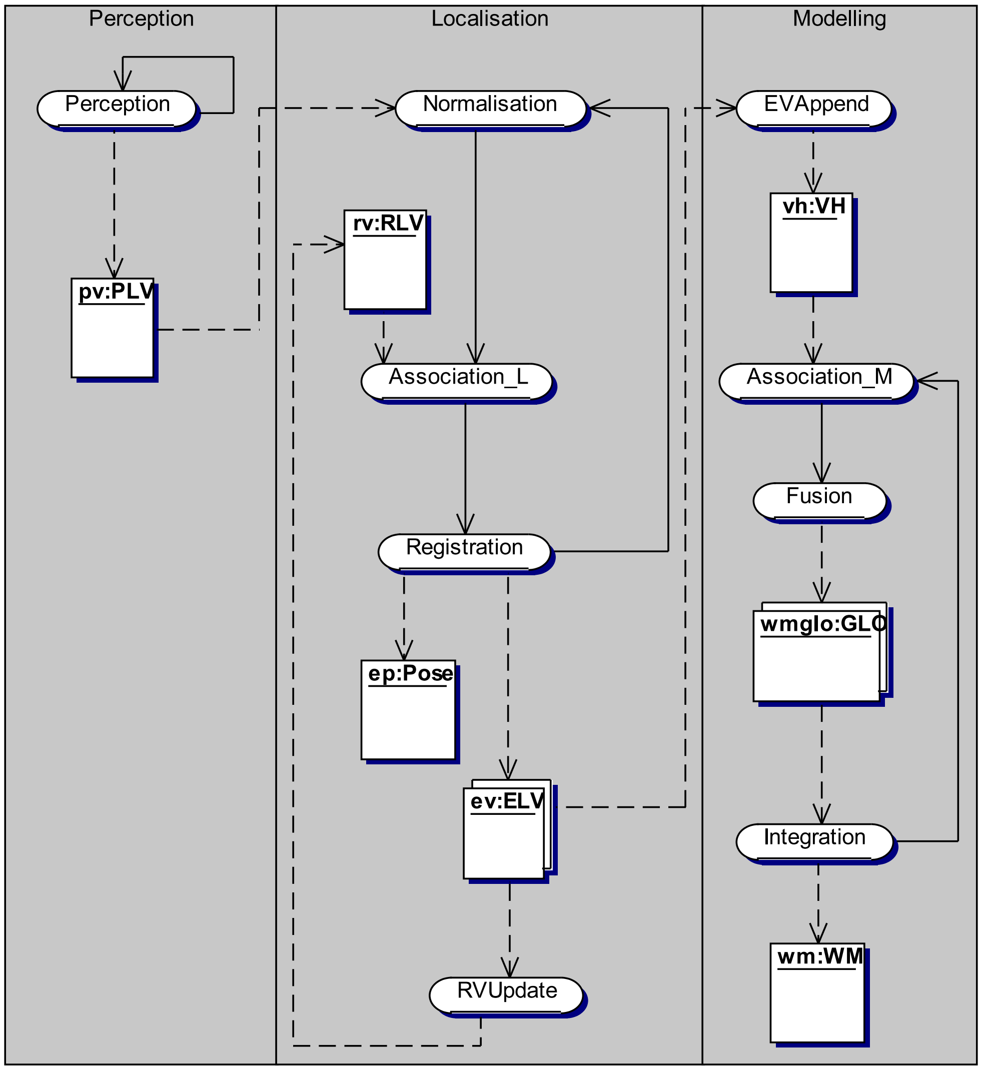


Figure 3.5: The activities concurrency

After the Association_L sub-activity completion, the transition toward Registration sub-activity may occur. Registration sub-activity produces an *estimated located view* (ev:ELV). The occurrence of transition to Normalisation sub-activity specifies that a new *Localisation* activity may be executed.

The dashed arrow exiting from Registration and connecting ev:ELV plays a crucial role, i.e.:

1. ev:ELV is not a single box, but a set of boxes, i.e., set of *estimated located views*;

2. ev:ELV represents the information that *Modelling* activity exploits (dashed arrow connecting ev:ELV to its EVAppend sub-activity).

The first consideration emphasises that the two activities have different execution rates: *Modelling* may exploit more than one *estimated located view*. This means that the execution rate of *Localisation* is greater then the *Modelling* one.

The second consideration enphasises that *Modelling* exploits a set of *estimated located views* ELVs, contrarily, *Localisation* exploits a *perceived located view* (pv:PLV). This difference is at the basis of the concurrence between the activities. Even if both PLV and ELV represent a located view, these views are different both conceptually and concretely. Conceptually, the first is a *perceived located view*, whereas the second an *estimated* one; concretely, we have two well distinct objects (one created from `PLV` class, whereas the second from `ELV` class). This consideration enforces the fact that no information is updated by more that one activity.

The last sub-activity of *Localisation* is the RVUpdate one. Its purpose is to provide Association_L the reference view RV.

Concerning *Modelling* activity, the set of *estimated located views* are each other linked to realise a *history of estimated views* VH. This operation is performed by the EVAppend sub-activity.

Successively, the *history of estimated views* VH is exploited by the Association_M (reifying the Association step in *Modelling* activity) to correlate sets of geometrically located objects.

After the completion of the Association_M sub-activity, its transition toward Fusion sub-activity occurs.

Finally, after performing Fusion and Integration sub-activities, an updated *world model* (wm:WM in the diagram) will be available.

3.4 Criticalities

As presented both in section 3.2 and 3.3, criticalities arise whenever the loop is closed, i.e., when a syncronisation between *Localisation* and *Modelling* activities is needed. Criticalities occur when:

- the *reference located view* RLV is not a plausible representation of the environment at the moment the *perceived located view* PLV was captured;

- the *history of views* VH is not a plausible representation of a suitable *world model*.

First criticality involves the *Localisation* activity, whereas the second, the *Modelling* one.

3.4.1 Management of Localisation criticality

A criticality occurring during the execution of *Localisation* activity implies that the *reference located view RLV* is not plausible. What we mean is that *glos* belonging to current *perceived located view* can not be matched against *glos* belonging to the *reference located view*. This implies that the *reference located view* must be replaced with another one that might describe the environment (with respect to the current *perceived located view* PLV).

Referring to figure 3.5, a criticality arises when the transition between Association_L sub-activity and the Registration one is not resolved. Indeed, it is exactly at this point of *Localisation* that the plausibility of the *reference located view* RLV is evaluated. At this aim, the transition is a *guarded* one. A guarded transition is a transition that may occur if and only if its precondition is satisfied (i.e., the guard resolves to "true"). In our case, the transition may occur if and only if the *reference located view RlV* is matchable against the perceived one PLV. If this condition is satisfied, then *Localisation* activity may go on (i.e., the Registration sub-activity is reached). On the contrary, a criticality occurs.

To overcame the criticality, a plausible *reference located view* may be derived from the *world model* WM. Two situation may occur:

1. the *world model* exists and it is a plausible reference view;

2. at the moment in which criticality occurs, the *world model* still does not exist.

CLAM successfully faces both the two situations: the first is solved by properly synchronising the information exchange between *Localisation* and *Modelling* activities; the second is solved by also adjusting the relative rates of the activities.

Concerning the first failure, we may again exploit an activity diagram to explain the "recovery" strategy. At this aim, figure 3.6 sketches the activity diagram modelling the synchronisation of the *Localisation* activity with the *Modelling* one. The diagram is equal to the one presented in figure 3.5 excepts that for the following:

- the presence of a *branch* connected to the transition exiting from Association_L sub-activity. The branch has two guarded outgoing transitions: only one of them can be taken. If "PLV matches RLV", then we have the concurrence has described in the previous section. If this condition is not satisfied, than this means criticality;
- the presence of the dashed line connecting the *world model* (wm:WM) to the RVUpdate sub-activity. The object flow introduced specifies the synchronisation between the two activity: they now share the very same information reified by the *world model*. The *world model* is provided to the RVUpdate sub-activity, so that it may use it to update the *reference located view* (rv:RLV);
- the presence of a new transition from RVUpdated to Normalisation sub-activity. When the *world model* is retrieved, *Localisation* can perform again its normal (concurrence) execution;
- the absence of the dashed line connecting the *estimated located view*s (ev:ELV) with the RVUpdate sub-activity. Since RVUpdate exploits the *world model* to update the *reference located view*, it does not exploit the *estimated located view* (ev:ELV).

We have closed the loop: transition outgoing from Association_L enter in a branch, the outgoing selected transition is toward RVUpdate, RVUpdate retrieves the world model to update the reference view, then its transition toward Normalisation is resolved.

After the execution of Normalisation and Association_L, again the branch is reached. If the condition "PlV matches RlV" is evaluated true, then the synchronisation between the activities successfully faced the criticality and the activities may be again executed concurrently. On the contrary, we still have the problem, even if this criticality is different. Indeed, either of the following situations occurred: the *world model* did not satisfied the plausibility constraint[12] as required, no *world model* was available (second pointed criticality).

Both the situations may be solved by adjusting the relative rates of the two activity. In detail, *Localisation* rate must be slowed down, whereas the *Modelling* rate must be speeded up, so that an updated *world model* will be available as soon as possible. In addition, the rates should be adjusted to achieve the following result: RVUpdate sub-activity execution will be performed when world model is ready and already provided to the RVUpdate sub-activity.

When the *Localisation* still fails, then a change of high-level strategy is needed, but is out of the scope.

[12] *Glos* belonging to current *perceived located view* can not be matched against *glos* belonging to the *world model*

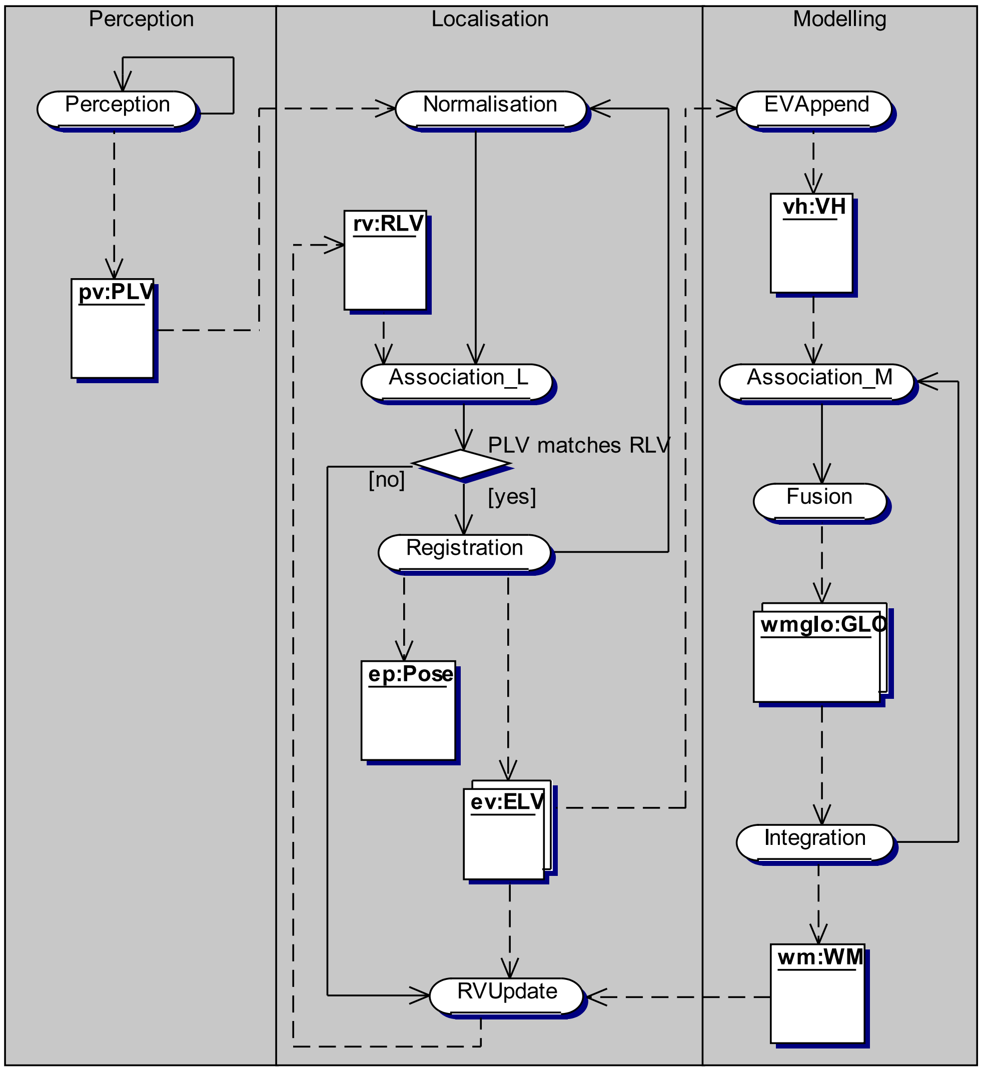


Figure 3.6: The activities synchronisation

3.4.2 Management of Modelling criticality

A criticality occurring during the execution of *Modelling* activity implies that the *history of views* VH is not plausible. What we mean is that corresponding *glo* objects cannot be identified in the recent history of view HV. Such a history does not provide useful information for *Modelling* purpose.

This criticality may be overcome by updating the *history of views* VH.

Referring to figure 3.5, a criticality arises when the transition between Association_M sub-and the Fusion sub-activity is not resolved. Indeed, it is exactly at this point of *Modelling* that the plausibility of the *history of views* HV is evaluated. At this aim, the transition is a guarded one: the transition may occur if and only if sets containing corresponding *glo* objects exist in the *history of views* HV. If this condition is satisfied, then *Modelling* activity may go on (i.e., the Fusion sub-activity is reached). On the contrary, a criticality occurs.

CLAM approach may solve the criticality by adjusting the relative rates of the activities. Precisely, *Modelling* activity rate must be slowed down, whereas the *Localisation* rate must be speeded up. The rate must be adjusted to achieve the following result: new Association_M sub-activity execution will be performed when an updated *history of views* HV will be available.

Even if this "recovery" strategy goes wrong, then a change of high-level strategy is needed, but, likewise *Localisation*, is out of the scope.

3.5 Timing

The issue concerning timing deserve some more discussion even if has been touched previously concerning criticalities.

The timing of the *Localisation* and *Modelling* activities are different and independent each other. Their independence is a consequence of concurrence, whereas their difference is due to the different lifetime of the information they produce. In the following we are interested in this peculiarity.

The *estimated robot pose* EP is an information the lifetime of which (i.e., existence period) is short. As a matter of the fact, the robot, to explore the environment, continuously move inside it. Independently from the dimension of the robot displacement, each movement causes the execution of a new *Localisation* activity (since the robot must understand which is its pose after the movement). Consequently, the new *estimated pose* replace the previous value.

Contrarily, the *world model* WM is an information the lifetime of which is long. The robot, when exploring an unknown environment, moves slowly both to avoid obstacles and to understand its pose with respect to the environment. Small robot displacements let the environment the robot perceives still be quite always the same, i.e., constant. This means that the

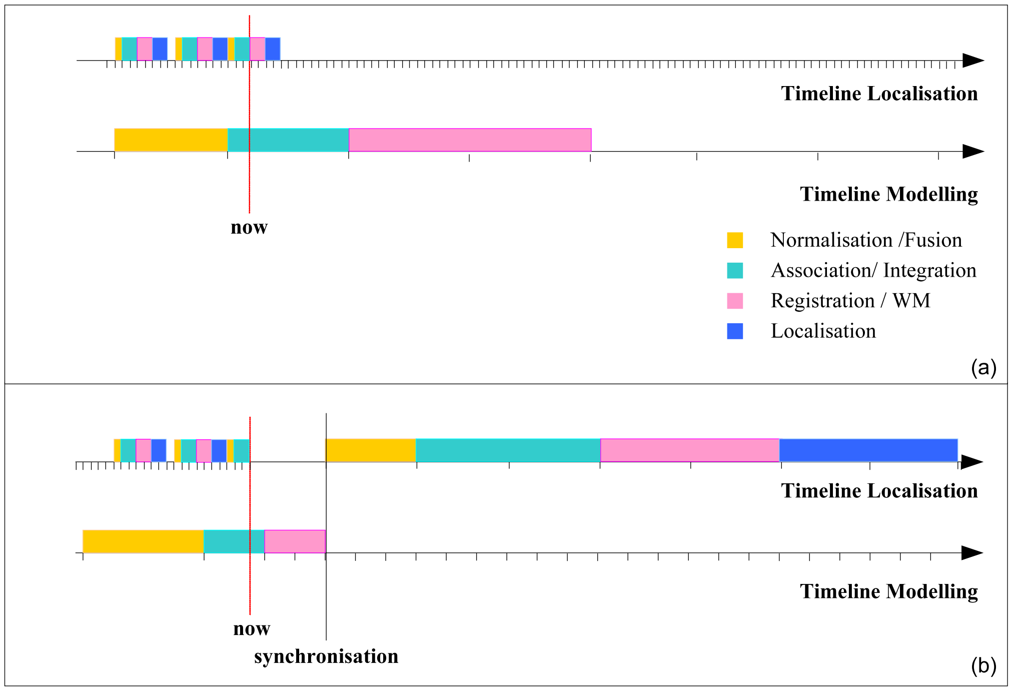


Figure 3.7: The activities timing

world model changes (better say evolves) very slowly. This consideration is enforced when the environment is structured and immutable (at least for a long term).

The above considerations justify that the activities are characterised by different timings. Precisely, *Localisation* must be executed with a greater rate than the *Modelling* one.

Let use imagine to model time as a line. The line is marked with ticks representing a time scale. The more tightened are the ticks on the line, the faster time passes. Activities steps are placed over a timeline. In doing so, both their starting execution time and their duration is explicitly defined. Under this model, a timeline completely describes the temporal behaviour of the activity.

In a CLAM system we need two timelines: one modelling the temporal behaviour of *Localisation*, the other of *Modelling*. Since the execution rate of *Localisation* is faster than *Modelling*, the time scale of *Localisation* timeline is finer than the *Modelling* one (see figure 3.7.a).

By opportunely enlarging or reducing the time scale of each timelines, it is possible to control the execution speed of the CLAM activities. This leads to face the criticalities. Figure 3.7.b shows how the criticality concerning *Localisation* may be faced: if the *world model* does not exists, or it is not a plausible representation of the environment, then, the time scale of *Localisation* is enlarged, whereas the time scale of *Modelling* is reduced. This lead to modify the execution speed of the activities in order to synchronise them. The figure shows how a proper selection of the activities execution speeds allow to obtain that *Modelling* ends just in time for a new Normalisation step of *Localisation*.

It is interesting to note the effects produced by playing with different execution rates, e.g., the enlargement of the time scale of *Localisation* may cause the robot to slow down.

The above described concepts will be treated in detail in chapter 4 where the reference architecture Real-Time Performers (RTP) will be introduced (another result of this work). Indeed, the RTP architecture supports the design (and implementation) of time sensitive system allowing the explicit control of the system temporal behaviour.

Finally, chapter 7 will present the exploitation of Real-Time Performers architecture for a real CLAM implementation.

Chapter 4

Real-Time Performers

4.1 Introduction

[1]

Software engineering usually focuses on methods, languages, and tools to specify what the software has to do in a strictly functional sense. Given a set of functional requirements, a number of methodologies, frameworks, and tools support the development of an application from architecture through design to implementation. The specification of a realistic system, however, will also include qualities like reliability, portability, security, distribution, timing, performance, and others desiderata. In the current practice of software engineering these qualities are often neglected, and classified as non-functional requirements.

In general, treating at architectural level non-functional requirements as thoroughly as functional ones is a difficult task. On the one hand architecture aims to abstract from details of implementation ([35] and [83]), but on the other hand many of the additional qualities required to the system restrict the realisation of its functional requirements and largely depend on their implementation.

Usually, the non-functional requirements that elude proper treatment at the architectural level are somehow dealt with at the design and implementation stages. Common techniques span from introducing special-purpose components (middleware) to tinkering with platform-dependent details.

This is especially true for the class of time-sensitive systems. We consider a system *time-sensitive* if it has to perform a number of different activities with multiple, dynamic, and inter-dependent temporal requirements which include, among others, timeliness, predictability, Quality of Service (QoS), performance, and so on. Furthermore, a time-sensitive system must be able to dynamically change both its activities and their temporal requirements. Actually, it should adapt itself to the intrinsic nature and variable amount

[1]The content of this section was elaborated with Sergio Ruocco and Andrea Trentini

of processed data, to the unpredictable and exceptional situations it may face, and to the varying performance and limitations of physical resources instrumental to its computations.

Time-sensitive systems include properly the set of real-time systems "whose correctness depends not only on the logical results of the computation but also on the time at which the results are produced." [110]. In addition, their correctness may depend also on performance and power consumption, which are temporal aspects of computations often considered aside to the domain of real-time systems. Hence the definition of time-sensitive systems.

We argue that the principles underlying current approaches to design of time-sensitive systems are not adequate to address the temporal aspects of computations at architectural level. We identify the main issue in the lack of proper architectural concepts that match the generality and the scalability of those employed with success in modelling the behaviour of a system in a purely functional sense.

To address this problem we introduce a novel approach to the architectural design of time-sensitive systems based on *computational reflection* [104] (see section 2.4). We model the temporal behaviour of a computational system with a set of architectural abstractions and, upon these bases, define *Temporal Reflection*, a new class of reflection, as "the ability of a system to self-represent, observe, and control its own temporal behaviour".

The idea behind this novel approach have been reified in a software architecture providing the basis upon which time-sensitive system my be designed. Within the architecture, we have developed a framework that may be used when building a system according to our architecture.

Even if major contributions of this work concern the software architecture (presented in section 4.2) and the related framework (presented in chapter 5), in the following section a brief explanation about Temporal Reflection will provided since it is at the basis of the architecture.

4.1.1 The underlying idea: *Temporal Reflection*

Since we argue that current approaches to design of time-sensitive systems do not address the temporal aspects of computations at architectural level, we identify specific architectural abstractions that properly reify the temporal behaviour of the system. Those abstractions are presented in figure 4.1.

A *RealTimeLine* reifies the "real" time, i.e., a monotonic sequence of time instants characterised by the non-decreasing `now()` value of the current time. The current time splits the timeline into a past and a future timeline.

An *Action* reifies the temporal aspects of a computational operation. It associates a `perform()` operation with a *TimeInterval*.

A TimeInterval defines two <begin, end> instant pairs modelling the planned time interval and the actual time interval for the operation execu-

```
/** Time Interval abstraction */
TimeInterval {
    int plannedBegin, plannedEnd;
    int actualBegin, actualEnd;
    TimeInterval(int pBegin, int pEnd);
}

/** Action abstraction */
abstract Action {
    TimeInterval timeInterval;
    abstract void perform();
    Action(TimeInterval tInt){
        timeInterval = tInt;
    }
}

/** RealTimeLine abstraction */
RealTimeLine {
    void addAction(Action);
    int now();
}
```

Figure 4.1: Temporal Abstractions

tion. Actual instants of a TimeInterval make sense for the past timeline only. They allow the past temporal behaviour of the system to be *recorded and observed*. Both actual and planned instants are immutable for the past timeline. On the other side, Actions can be inserted into the future timeline and the planned instants of their TimeIntervals can be specified. This allows the global temporal behaviour of the system to be *controlled* by properly planning actions.

An execution engine triggers the execution of a computational operation whenever the corresponding action is enabled, i.e., the current time falls inside its planned interval, and sets the actual begin and end of the interval whenever a computational operation is actually started and completed respectively. The execution engine provides the required reflective causal connection between the actual system temporal behaviour and the architectural abstractions that represents it, i.e. the actions and the real-time line.

Finally, a Strategist is in charge of observing the past behaviour of the system and of planning its future behaviour by observing Actions in the past timeline and by setting Actions in the future timeline according to the application goal(s) and requirements. Upon these bases it is then possibile for a computational system to perform a new kind of reflection, that is *to reason at architectural level about temporal aspects of its own computations.* Accordingly, we define *Temporal Reflection* in general as "the ability of a system to self-represent (reify), observe and control its own temporal behaviour".

4.2 A Reference Architecture for Real-Time Systems

An application program to explicitly deal with non-functional requirements needs abstractions capturing timing and architectural issues to allow the observation and the control of both the internal architecture and the temporal behaviour of the system at the application programming level. *Real-Time Performers* (RTP in the sequel) tries to capture and to model these abstractions into a *reference software architecture* for the design of modular, distributed, and real-time systems.

As explained in section 2.3, it is widely known that there is a little agreement on the definition of software architecture. Thus, when talking about architecture, one must make it clear what he actually mean. Basing upon the definitions given in [102] and [35], in our point of view, software architecture comprises:

- the overall organisation of components and connectors;
- the externally visible properties of components;
- the overall system strategy.

A *reference software architecture* is a special kind of software architecture. Unlike "concrete" architectures (i.e., architectures of actual, running systems), a reference architecture must be general enough to accommodate a wide range of concrete architectures, yet provide software architects and/or designers with the basic conceptual building blocks for easily developing systems in a certain application domain. In other words, a reference architecture should contain the vocabulary of the domain i.e., the basic concepts found in the application domain, and a large fraction of the knowledge required to solve a composite problem [117].

4.2.1 Background Projects

RTP is the result of previous research activities. In particular, works like HyperReal ([86], [89], [22], [23], and [90]), Kaleidoscope ([94], [96], and [95]), and RAID[2] ([80], [77], and [71]) have strongly influenced the design of RTP architecture.

[2]Rilevamento dati Ambientali con Interfaccia DECT

HyperReal is a software architecture focusing on time issues, Kaleidoscope is a software architecture designed for monitoring and control systems, and RAID is a monitoring and control system for Indoor Air Quality (IAQ) within ancient buildings.

HyperReal focuses on timing issues, defining an entity (named *controller*) that relies on a time-driven model of control, and separates the definition of plans from the dispatching of actions they define. Special entities named *virtual clock* support the explicit management of time.

Kaleidoscope architecture relies upon a general mechanism for the definition of software components and their composition, and maintains at the application level the definition of the policies controlling information exchange. Kaleidoscope captures the common characteristics shared by almost every monitoring and control systems from the point of view both of the application domain model and of the architectural requirements. From the domain model point of view, these systems deal with physical entities represented at different levels of abstraction. From the architectural point of view, these systems must ensure reusability: computation (data elaboration), distribution (data dissemination), and activation (data distribution and elaboration policies) must be strictly separated. Kaleidoscope relies on abstract representations of the domain entities (*conceptual images*) that are mapped into concrete representations (*concrete images*) hosted by heterogeneous components. Connectors (*projectors*) allow different representations of the same abstract entity to be aligned according to domain-dependent strategies. Both concrete images and projectors do not encapsulate any policy about *when* processing and alignment are done. The system overall control is assigned to *strategists*: the entities responsible for activating image alignment and information processing.

RAID is the result of a three-year project (ended in December 2002) funded by the Italian Ministry for University and Research. The RAID project goal is to design, develop and deliver an IAQ monitoring and control system devoted to buildings characterised by the absence of automatic control devices (e.g., fire and microclimate control plants). Wireless communication (due to a possible lack of cabling), innovative sensors for environmental measurements, and the capability of inferring pollutant sources are the main features of the project. The system acquires data from microclimatic, pollutant, and radon devices. Since the RAID target buildings may not host cables, an acquisition module based on wireless network (DECT) has been developed. It acquires measurements and distributes them to the Main System through a TCP/IP infrastructure. Within the RAID project the knowledge-based approach has been followed in order to support human experts in IAQ control. In particular, the SER (Sistema Esperto RAID) application has been developed in order to identify possible related causes (i.e., pollutant source localization) when IAQ anomalies are detected. Beside the SER module, specific applications have been developed for RAID purposes:

a specific configuration module, a data mining application, a 3D graphical user interface, and a neural network. Due to their heterogeneous nature, each of these modules needs specific information arranged in a suitable way. Their integration should have been a challenge without a specific underlying software architecture. Thus, the RAID system has been designed according to Kaleidoscope principles.

4.2.2 An Overview of RTP

In our opinion, both *computational* and *architectural* reflection (see section 2.4) are the starting points to build dynamic systems. Computational reflection is defined as the activity performed by an agent when doing computations about itself [70]. Whereas, architectural reflection [15] is the computation performed by a system about its own internal architecture. It reifies architectural features as meta-objects which can be observed and controlled at runtime. The application of architectural reflection helps bringing visibility over the computation performed by the *overall* system components at the programming level. Finally, temporal reflection (see subsection 4.1.1) adds to system the capability in reifying, observing, and controlling its own temporal behaviour.

Reflection principles have guided the modelling of RTP architecture in order to raise at the application programming level:

- strategies definition (action choice on behalf of events);
- timing issues management (speed-up/slow-down tuning);
- component behaviours definition (adding/removing performable commands);
- system topology definition (adding/removing connected components).

RTP is based on the following key concepts:

- a system is made up by computational components;
- computational components exchange information via alignment components;
- both computational and alignment components are activated and controlled by supervising components.

The above three key concepts have lead to individuate three well distinct corresponding roles inside a system: the role of *computing data*, the role of *distributing information*, and the role of *activating both computation and distribution*.

RTP assigns these roles to three specific components: *Performer*, *Projector*, and *Strategist*. Performer is the entity expressly designed to *perform* elaboration on own data, Projector is the entity expressly designed to *project* (distribute) data between Performers, and Strategist is the entity expressly designed to device or to employ plans or *stratagems* toward goals like the activation of Performer computations and Projector alignments. The identification of these well-distinguished components emphasises the "separation of concerns" between information processing, information alignment, and their activation.

To achieve reuse both off-line and on-line, the following requirements must hold:

- Performer should be unaware about both the surrounding environment (in terms of topology) and the system behaviour; it should perform its activities only upon triggered commands;
- Projector should be aware about only the Performers it projects, but it should be unaware about the system behaviour; it should perform alignment activities only upon triggered commands;
- Strategist should be aware about the overall system behaviour. Strategies should be dynamically created thanks to reflective mechanisms.

RTP capture all of the above requirements in a reference architecture that may be exploited for the design of strongly modular, distributed, dynamic, and real-time systems. Its main peculiarity consists in the ability to arise at the application programming level both timing and architectural issues allowing the changes of behavioural strategies dynamically. Moreover, RTP allows strategy changes without affecting components implementation. Since strategies can be planned at the application domain level instead of being embedded in component, RTP allows application to easily satisfy non-functional requirements.

Summarising, RTP may be considered as a reference architecture for those systems in which the observation and the control of the temporal overall system behaviour is a key issue.

RTP architecture has been positively accepted by the scientific community. Indeed, RTP has been published in the following referred papers:

- [79] is the first published paper about RTP. In this paper we present a mapping between a conceptual architecture based on a multi-agent approach for Computer Supported Cooperative Work (CSCW), and the RTP architecture;
- [82] is the first stand alone description of RTP concepts and idea. The focus is on real-time data distribution;
- [81] enphasises the role of Performers and the dynamical change of components behaviour;

- [78] presents a proposal for RTP exploitation in a complex distributed movement tracking system based on webcams.

An object-oriented approach, based upon the Unified Modelling Language (UML) (for detail the reader can refer to [43], or the more exhaustive reference [93]), has been employed throughout RTP lifecycle. Therefore, the UML will be used in the sequel for documentation purposes.

4.3 Performers

A Performer is the entity executing specific activities useful for the system, e.g., converting an image, calculating a pollutant concentration, and so on.

Performer may require data to execute its tasks, and the performed activities may produce data.

Since a Performer must be a passive entity (to not encapsulate any kind of activation strategies), it must be activated in order to perform activities. At this aim, it must be able to accept a suitable set of commands. The execution of an accepted command may influence the Performer state. These considerations lead to model a Performer as a classical state machine:

- transitions correspond to the execution of local transition functions (i.e., actions, or, methods adopting an O-O parlance);
- the execution of a transition is atomic;
- the definition of a local transition function does not include any reference to timing.

The definition above can be refined to model two abstraction levels that are relevant in terms of both model and architecture. Imagine a Performer with two sets of states: one defined *macro*, and the other *micro*. The set of macro states corresponds to the states in which a Performer specifies to be after receiving a command (i.e., the state that a Performer decides to *export*). The set of micro states corresponds to the local *environment* of a Performer, i.e., a set of local variables defined inside the Performer only.

The change of Performer macro state is due to a transition that is triggered by a received command. Transitions are associated with actions. The right transition is selected accordingly to the received command and a guard[3]. A guard is a boolean expression depending on the local environment. Guards are expressed in terms of environment variables. Actions manipulate environment variables. This means that the (macro) state evolution is not affected by the performed actions (at least for a single transition).

Adopting a formal notation:

[3]If more than one guard is true, the selection of the transition is nondeterministic. If the modeling of a fully deterministic system is required, there should not be non-deterministic guards (or the non-deterministic choices should not affect the overall system behavior).

- a Performer i is a t-uple $P_i = \{S_i, s_i, T_i, E_i, G_i, A_i\}$, where:
  - S_i is the set of states;
  - s_i is the initial state;
  - T_i is a set of transitions;
  - C_i is a set of commands;
  - E_i is the local environment, i.e., a set of variables in a local name space;
  - G_i is a set of guards defined over E_i;
  - A_i is a set of actions maipulating E_i.
- a transition $t \in T_i$ is defined as $t_i = \{s_o, s_d, c, g, a\}$, where:
  - $s_o \in S_i$ is the origin state;
  - $s_d \in S_i$ is the destination state;
  - $c \in C_i$ is a command;
  - $g \in G_i$ is a boolean expression defined over E_i;
  - $a \in A_i$ is an action.

4.3.1 Visibles

The Performer local environment deserves more discussion. Local environment is composed by two different sets of variables: the set of *local variable*s, visible inside the Performer only, and the set of *visible variable*s, visible also outside the Performer.

In turn, the set of visible variable is arranged in two subsets:

- the set of *exported visible*, variable writeable by the Performer and readable from the outside (i.e., which can be assigned from the inside and observed from the outside);
- the set of *imported visible*, variable readable by the Performer and writeable by the outside (i.e., which can be observed from the inside and assigned from the outside).

This formal separation is important because it allows a Performer to be not aware of the information source and target respectively: a Performer may only publish data on the exported visibles and read data from the imported visibles. Thus, the set of visibles represents the only "port" to the outside world of the Performer: communication between the Performer and the rest of the system must always pass through this set. This must be regarded as an advantage because makes Performers completely unaware of the system in which they work. This means that such a component may be composed under different topologies.

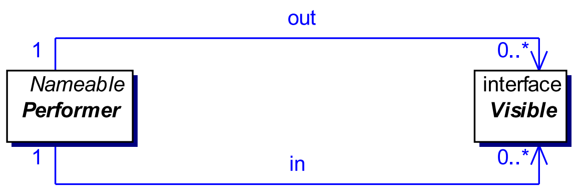


Figure 4.2: Performer and its Visibles

Finally, Performers have a local name space for their visible variable so that the visible variables cannot be shared explicitly. There are several reasons for this choice: first, it encourages design modularity; second, it helps separating local behaviour, topology and strategy; third, it properly models the concrete architecture of a loosely connected distributed system.

The UML diagram in figure 4.2 shows the relationship between a Performer and its (possibly) visibles.

Summarising, a Performer is a commands acceptor/executor providing data within exported visibles and requiring data within imported visibles. Its functioning is very simple:

- accept a command from the outside;
- decide if the command is acceptable;
- process the command by (optionally) changing state and (optionally) reading/writing visible variables.

4.4 Projectors

A Performer may require data produced by other Performers to carry out its activity. Moreover, a Performer may produce data useful to other Performers at the end of its computation.

Adopting a classical approach, a Performer should be aware about both the Performers producing the data it needs, and the Performers requiring data it produces. Such a solution implies Performers knowledge about the system topology unwavering the component reuse under different topologies.

To overcame this restriction, in RTP information exchange between Performers is possible only if data is *projected* between Performers. In other words, a Performer can observe the exported visible of another Performer only if the exported visibles are projected into its imported visibles. At this aim, RTP introduces the concept of Projection. A Projection is a pair (*source*, *target*) where *source* is an exported visible of one Performer, and *target* is an imported visible of another Performer.

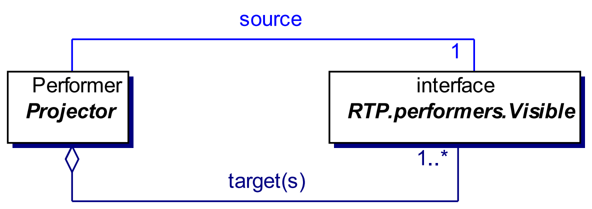


Figure 4.3: Projector

In terms of architecture, Projectors reify Projections. Thus, Projectors are the entities that allow data sharing between Performers defining mappings from visible exported variables of a Performer into imported visible variables of another Performer.

Defining Projectors, the system topology will be described without embedding inside the Performers the knowledge neither about the Projections nor of other Performers.

Projectors are privileged Performers that reason “in-the-large”:

- they have visibility over a global name space, where the visible variables of the Performer are identified by a pair (*performer*, *visible variable*);
- they have visibility over Projection definitions;

Projectors may connect different pairs of (*performer*, *visible variable*) during system lifecycle. Since Projections are reified as concrete objects, the capability of manipulating them is the basis for topological reflection, i.e., the capability of changing the system topology according to the observation of its own state.

Finally, the definition of Projectors not imply any assumption neither about *why* and *when* observation is done, nor about observation mechanisms (polling, asynchronous messages, and so on) whose definition is a matter of system dynamics not of topology.

The UML diagram in figure 4.3 shows the relationship between a Projector and the visibles that it aligns.

4.5 System dynamics

Performers and Projections are not aware about system dynamics: information about *when* and *why* an elaboration and an alignment occur is not in charge of these kinds of entities.

The definition of system behaviour is assigned to a specific entity that has the complete view of both the system topology and the system condition. By *Strategy* we mean all the issues related to *when* and *why* Performers and

Projectors are triggered to compute and align respectively. This approach has the goal of maximising reuse of both Performers and Projectors under different dynamics. Strategy defines the future system behaviour observing the current system state, the past system state, and the future planned behaviour.

The Strategist (the reification of Strategy) is conceptually a single unit. In practice, it can have any implementation, from an actually single entity (for very simple, centralised systems) to a complex, distributed system, made up of a number of clocks, schedulers, etc. In other words, it is not our intention to centralise control (with all the known drawbacks) but simply to separate control from the other two orthogonal issues (namely, computation and distribution).

When an action has to be executed, depends on the kind of system we are realising: in system where time does not play a crucial role, an action may be executed, e.g., after the end of another one; whereas in real-time system, an action must be executed within a specific time interval.

Even if RTP has been designed for real-time system (see subsection 4.5.2), it may be exploited also for other kind of systems with no timing constraints. This is an appreciable result deriving from the separation of concerns that RTP operates on control, computation, and distribution.

4.5.1 Systems not dealing with time issues

In its simple definition, a *Trace* defines a part of the system behaviour in terms of partially ordered set of *Requests*. A Request specifies only the *recipient* and the *action* to be performed without any information about the time in which the Request has to be delivered.

Under the condition of nondeterministic guards, all the Traces defined by a Strategist describe the overall system behaviour. The proposed approach allows a fully deterministic behaviour to be defined, if required by the application domain. If a fully deterministic behaviour is not required, nondeterminism simplifies the specifications by avoiding unnecessary constraints.

Traces defined for a system are arranged into two subsets: one containing the Requests that have already been delivered (Past Trace) and the other containing the Requests that have to be delivered (Future Trace).

Past trace represents a log of the system behaviour in terms of what has happened. While, Future trace represents the expected system behaviour in the future. A Strategist may change only dynamically this future behaviour by adding, removing, or moving Requests.

4.5.2 Systems dealing with time issues

The RTP complete exploitation occurs when designing and implementing real-time systems. Indeed, RTP manages timing basing upon Temporal Reflection: temporal abstractions identified in subsection 4.1.1 are at the basis of RTP timing management.

The basic concepts about time in RTP are the following:

- *time* is discrete, i.e., time instants are modelled by nonnegative integers;
- a *timeline* is a monotonic sequence of integers;
- *current time* represents a point in a timeline, and is bound to change monotonically.

When introducing timing issues, Trace and Request have to be modelled consequently, i.e., they must be fleshed out with time information. Consequently, we define *TimedTrace* as a set of *TimedRequests*, i.e., pairs (*request*, *interval*) where interval is a segment of the timeline, defining its execution planned time. Referring to the temporal abstractions defined in 4.1.1, TimedTrace reifies the RealTimeLine, TimedRequest the Action, and interval the TimeInterval.

Note that associating Requests with intervals implies a partial ordering of the requests. TimedRequests are only eligible for being executed when current time falls inside the associated interval.

A TimedTrace keeps track of current time exploiting *Virtual Clocks*. A virtual clock is an active component that is in charge of advancing the current time of a timeline (for detail see [86] and [89]). A tick is an increment of the current time of a timeline.

A virtual clock is associated to one reference clock (which, in turn, is a virtual clock). The period of a virtual clock is the number of ticks of the reference clock for each tick of the virtual clock. Informally, the period of a virtual clock defines the timeline advancement speed. In general, several virtual clocks may have the same reference clock. Therefore virtual clocks are logically arranged in a forest. When we need a system-wide clock the forest collapses to a tree whose root is the system clock. It is a clock whose advancement is driven by events that do not fall inside the model (for instance, physical interrupt or keystrokes or simulation events).

TimedTraces rely on virtual clocks; therefore they are not aware of any kind of “absolute” time. The “absolute” execution speed of the TimedTrace defining the behaviour of a sub-system can be controlled from the outside. Of course, this holds for the future portion of the TimedTrace. The past portion of the TimedTrace, including the past periods of the virtual clocks, cannot be changed. A straightforward architectural mechanism is to define the past portion of a TimedTrace according to a virtual clock whose period is immutable. In many cases, the system clock has an immutable period - or a period that can be assumed as immutable if compared with the “real” time.

Finally, when a TimedRequest is executed, it is updated with the additional information concerning its actual execution time, and from now, belongs to the past TimedTrace.

4.5.3 Strategy

A Strategist, as said in the beginning of section 4.5, is in charge of defining the system behaviour. At this aim, it defines the set of future Requests as a function of the current Trace and of the observable state of the Performers.

The model accommodates both on-line and off-line strategies[4]. Furthermore, the model assists different Strategies without affecting both Topology and Performers definitions.

For instance, an alignment can be triggered according to three different Strategies:

- *push*: the Strategist states that a Request is issued to a Projector when its source variable changes;
- *pull*: the Strategist defines that a request is issued to a Projector on a change of a suitable exported visible variable of the Performer the target belongs to;
- *timed*: the Strategist triggers the execution of actions according to a specific timing.

Similar remarks apply to the Strategies that generate actions to Performers.

Strategist may modify a Trace (the future portion) upon system observation. Mechanisms and policies used to observe the system must be confined into the Strategist itself: it may poll interesting visibles or it may ask visibles to generate notifying events.

Of course, you can devise a global Strategist controlling the overall behaviour. At the extreme, there might be a Strategist for each Performer (i.e., each Performer becomes an autonomous thread).

What is relevant is that the model allows strategy changes dynamically during the lifecycle of the system. In fact, since Strategies can be planned at the application domain level, the system behaviour may be easily changed to respond to the specific requirements by changing, adding, and removing Requests inside a Trace (the future portion). Moreover, a Strategist may even change, at the application level, the dynamics of the system in terms of

[4]Of course, off-line strategies do not depend on the current observable state of the Performers.

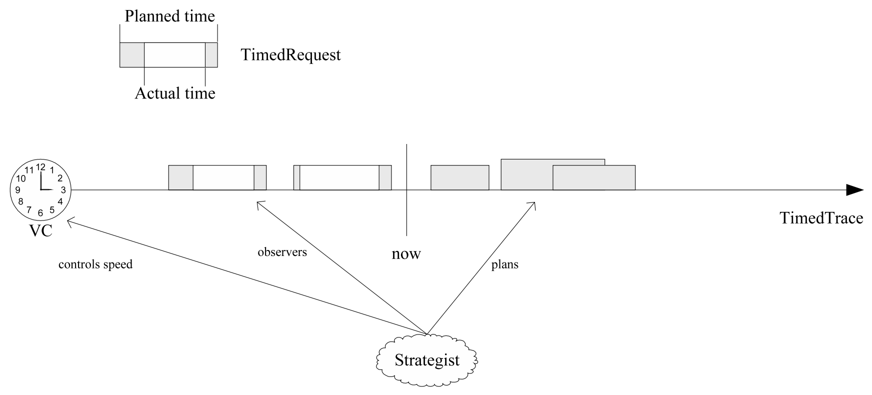


Figure 4.4: RTP system temporal behaviour controlled by a Strategist

timing issues, e.g., it may accelerate or decelerate some activities exploiting virtual clocks.

Figure 4.4 sketches the system temporal behaviour controlled by a Strategist as provide by RTP.

Chapter 5

Real-Time Performers Framework

5.1 Introduction

The chapter describes the substantial implementation work performed within Real-Time Performers. The reasons why we call it a framework are the following:

- the implementing work is completely independent by specific application domain. Under this respect, the resulting product may be considered as a class library;
- the designer may extend the class library by subclassing some of the existing classes;
- the presence of a configuration tool to support startup topology and initial strategy definition (under implementation).

The work presented in this chapter is therefore meant to be a common platform that a software engineer is free to use as it is, or modify (by specialisation of the basic provided mechanisms) when building a system according to the RTP architecture.

This chapter will show the program structure through detailed UML design diagrams, that have the advantage of being both high-level enough to be understandable and low-level enough for an experienced reader to grasp all the necessary details. In addition, whenever possible, we will not show class attributes and operations, so as to keep the diagrams as simple as possible.

Concerning the implementation language adopted for the development of the framework, our choice is fallen on the Java Programming Language [3] since it is an object-oriented language, it is independent from the specific platform, and it provides basic reflective mechanisms.

5.2 General Organisation

The framework has been logically structured into packages: each of them contains entities that are in someway related each other. Figure 5.1 shows the packages constituting the framework and the relationships among them.

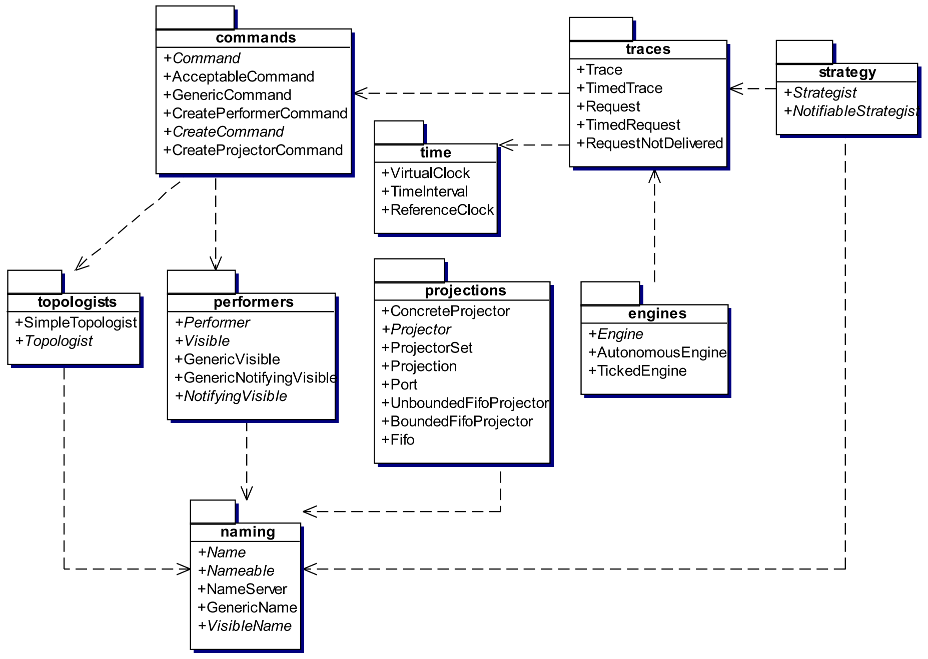


Figure 5.1: The RTP packages and their relationships

Even if each of the packages is described in detail in the next sections, a brief introduction will be provided:

- `naming` package deals with the uniqueness of the names for components in RTP;
- `performers` package contains all the classes useful to define the Performer entity as introduced in section 4.3;
- `projections` package deals with the Projector entity introduced in section 4.4;
- `topologists` package contains classes defining the Topologist entity (i.e., the entity that create Performers and Projectors);
- `commands` package defines the commands to be delivered in order to trigger actions;
- `traces` package reifies the dynamics of the system;

- `time` package deals with all the temporal aspects of the system;
- `strategy` package contains an abstract representation of the strategy;
- `engines` package deals with the distribution of commands among performers.

The rest of the chapter is organised in two sections: the first dealing with topological issues (section 5.3), and the second dealing with the management of of the system dynamics (section 5.4).

5.3 Topology

The packages described in this section contain all the issues related to system topology. Main concepts here are Performers and Projectors as defined in chapter 4.

Before entering into details about the reification of the RTP approach as described in chapter 4, the `naming` package will be introduced as first concept since it is present about in all the framework.

5.3.1 The `naming` package

Symbolic names are the preferred way, rather then using directly "references", to manage instances of objects inside a software system. Indeed, exploiting symbolic names, the distribution of references may avoided to preserve the consistence of the objects state.

When entities have to be referred uniquely inside the system, then naming is a prerequisite for the correctness of the system: the repository elected as manager of the correspondence $(name, reference)$ should also provides a mechanism that assure univocal names.

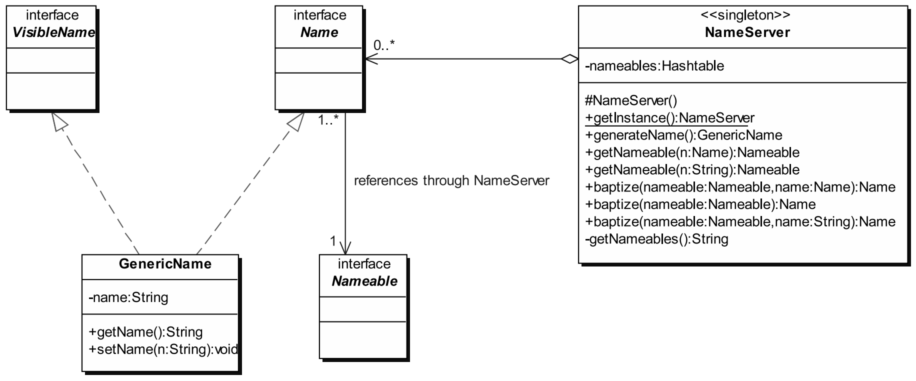

interface
VisibleName
interface
Name
0..*
<<singleton>>
NameServer
-nameables:Hashtable
#NameServer()
+getInstance():NameServer
+generateName():GenericName
+getNameable(n:Name):Nameable
+getNameable(n:String):Nameable
+baptize(nameable:Nameable,name:Name):Name
+baptize(nameable:Nameable):Name
+baptize(nameable:Nameable,name:String):Name
-getNameables():String
1..*
references through NameServer
1
GenericName
-name:String
+getName():String
+setName(n:String):void
interface
Nameable

Figure 5.2: The `naming` package

`naming` package aim is the management of pairs ($name, reference$) assuring the iniquity of names. The UML diagram of figure 5.2 shows the classes designed for this goal:

- `NameServer` class: is a class that serves to manage the references to all the nameable entities inside the system. There is exactly one instance of this class (stereotype << *singleton* >>). If the system is distributed, than there will be one instance for each node in the system. In practise, this class is made up by a hashtable that keeps track of the nameable entities through their symbolic names;
- `Name` interface: This is a tag interface representing a unique (global level) name that references a `Nameable`;
- `Nameable` interface: every entity inside the system that is someway referred must be identified using an unique Name. These kinds of entities must implement this interface;
- `VisibleName` interface: every Performer visible (see subsection 4.3.1 and 5.3.3) must be recognised by a name. This interface must be implemented by every class representing a name for a visible. A `VisibleName` could be a simple string;
- `GenericName` class: a simple implementation of both `VisibleName` (for visible entities) and `Name` (for `Nameable` entities). It simply wraps a string.

5.3.2 The `commands` package

This package, sketched in figure 5.3, contains basic entities representing the commands that Performers may execute.

`Command` interface and `AcceptableCommand` class are the core of this package. The former is a tag interface that must be implemented by every class representing a command. It is defined as tag interface since a command can be anything: from a simple string to a piece of software. Apart form its definition, a `Command` is meaningful *only* for a particular receiver.

`AcceptableCommand` class specifies if a command is eligible to be accepted by the receiver. Its `match(Command cmd)` method verifies the validity of the command `cmd` in input. This class is very useful to the command receivers since it encapsulates all the validity policy. This class may be subclasses in order to override as needed the `match()` method.

From this point of view, the `AcceptableCommand` class may be defined as a meta-representation of the performable commands by the computational components constituting the system.

The `GenericCommand` class is an example of a concrete (even if trivial) command. This implementation simply wraps a string.

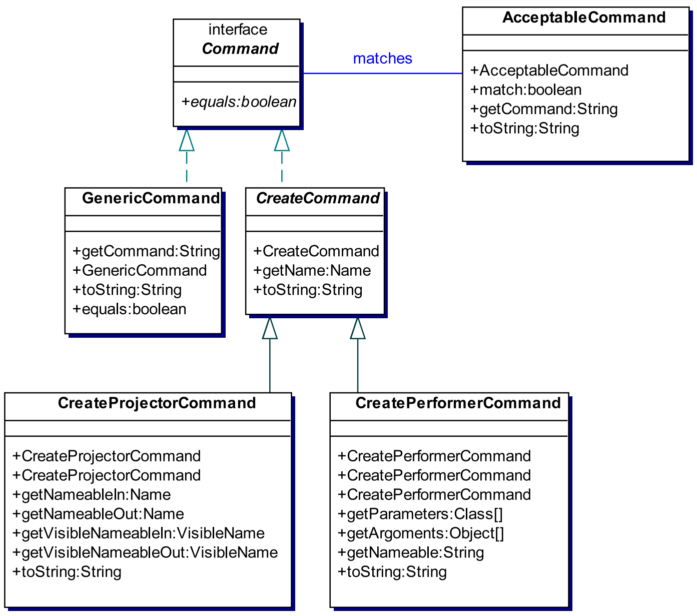


Figure 5.3: The `commands` package

Finally, class `CreateCommand`, and its concrete subclasses, define the set of commands useful to create (and destroy) dynamically Performers and Projectors.

5.3.3 The `performer` package

This packages, depicted in figure 5.4, contains all the classes representing the computational components described in section 4.3.

Main concepts here are the `Performer` class, reifying the Performer entity, and the `Visible` interface, reifying both the exported and the imported visibles of a Performer.

`Visible` interface represents both imported and exported visible variable whose handling is made using the `get()` and the `set()` methods respectively. For generality purpose, the methods respectively return and require `Object` instances (see code fragment in figure 5.5).

Concrete visible variables should implement the `Visible` interface. `Visible` implementations should model the data a Performer (possibly) needs to execute its task and the data it (possibly) produces for other Performers. An example of concrete implementation of Visible is the `GenericVisible` class: it is a simple wrapper for a generic object. This Visible definition is “semantic-unaware”, which means that `GenericVisible` instances represent data that is not related to specific application domain.

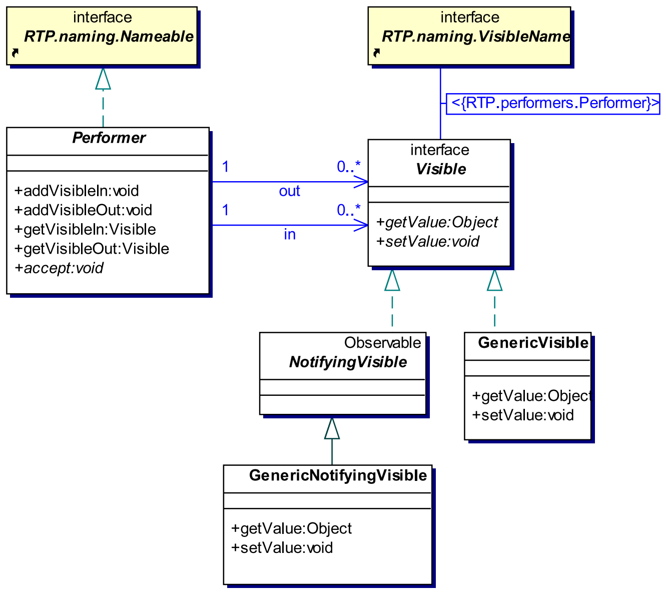


Figure 5.4: The `performer` package

Since the framework must be the most flexible as possible in order to satisfy the more wide choice between the possible observation mechanisms (e.g, polling, events, and so on), we introduce also a special kind of visible named `NotifyingVisible`. This visible is a specialisation of the `Visible` interface that sends notifications each time its value is updated. This class may be very useful when an event-driven policy is preferred to, for instance, a polling one.

`Performer` class models a Performer. It is a generic command acceptor with unique name (i.e., it is a `Nameable`). The uniqueness of its name is a fundamental constraint since its name is what will be used to identify the proper Performer when planning behaviour.

The double association with `Visible` interface (see figure 5.4) specifies respectively the exported (*out* label) and imported (*in* label) visibles of a Performer. From an implementation point of view, two separated hastables reify the double association. Since a Performer may be an independent entity (in the sense that does not need data to perform its activity and/or does not produce data from its computations), the associations with the `Visible` interface have cardinality $0..*$, instead of $1..*$.

The `accept(Command cmd)` method of `Performer` class is responsible for activating the Performer accordingly with the `Command` instance in input. `Command` represents a generic command to be executed by a `Performer` instance. Its semantic is local to the receiving Performer. Since at this level of abstraction, the definition of `Performer` class must fit with every kind

```
/**
 * A Visible exports get and set methods that should be used according to
 * a "protocol" (e.g., for a "in" value only the Performer can call the
 * getValue, ...)
 */
public interface Visible {
    /** Returns the value for this visible */
    public Object getValue();
    /** Sets the value for this visible */
    public void setValue(Object v);
}
```

Figure 5.5: The Visible interface definition

of concrete and domain-dependant implementation, its `accept()` method is defined abstract (i.e., not defined). Concrete Performers should be specialisations of the `Performer` class, and they have to provide an opportune implementation of the `accept` method definition.

From the considerations above, at this level of abstraction, the role of the `Performer` class is visibles management only. Code fragment in figure 5.6 exemplifies the concepts above described. This fragment emphasises the definition of methods for the management of Performer visibles.

5.3.3.1 The `programmableperformer` package

Another feature of RTP architecture is its support to the dynamical change of component behaviour. Indeed, RTP refines the paradigm of the Strategy Pattern [44] by applying the state machine model to the computational component with the purpose of dynamic behaviour change. By using reflective methods, the programmer (actually, the Strategist) may modify any part of the state machine, thus building a (potentially) completely different Performer than the one initially instantiated. The idea behind the choice of state machines (as inspiration for the computational component) is that there is a lot of research already available on FSA [53] (Finite State Automata) and their properties.

To create a truly dinamically (at runtime) changeable `Performer`, the `ProgrammablePerformer` class was defined as an extension. The `ProgrammablePerformer` represents an explicit state machine, it reifies states with `State` instances. Figure 5.7 shows the UML class diagram of this class with its set of states and ECA[1]-transitions. The `ProgrammablePerformer` has a set of `State`s, among them one is marked as "initial", and another as "current". Each `State` has a set of `Transition`s that will be examined when interpreting

[1]Event-Condition-Action

```
/**
 * A Performer accepts commands (automaton-like) and allows access to
 * visibles by name. The role of this level is Visible management only.
 */
public abstract class Performer implements Nameable {
    /** The imported Visibles */
    private Hashtable in;
    /** The exported Visibles */
    private Hashtable out;

    /** Adds an imported Visible */
    public void addVisibleIn(Visible v, VisibleName n) {
        in.put(n, v);
    }
    /** Every Performer gives a name to every Visible */
    public Visible getVisibleIn(VisibleName name) {
        return (Visible)in.get(name);
    }
    //... same methods for exported Visibles

    /** Classes must be specialised to implement the appropriate accept policy */
    public abstract void accept(Command c);

    //...
}
```

Figure 5.6: The Performer abstract class

acceptable commands. Each `Transition` reifies a ECA transition, i.e., it is associated with an (optional) `AcceptableCommand`, a (optional) `Condition`, a (optional) `Action`, and a (required) next `State`. The `accept()` method on the `State` class cycles through all the associated `Transition`s until the following two conditions hold: a `Transition` that matches the actual received `Command` exists, and the (optionally) associated `Condition` is evaluated as true.

If such a `Transition` is found, its corresponding `Action` is executed and its "next" `State` becomes the "current" one. Both `Condition` and `Action` need a link to the Performer (marked "context" in the figure 5.7) because they respectively need "read"[2] and "read and write"[3] access to the Performer.

In this model, each class reifies an aspect of the formal state machine described in section 4.3:

[2]A condition is evaluated against the state of the Performer.
[3]An action may also modify the Performer.

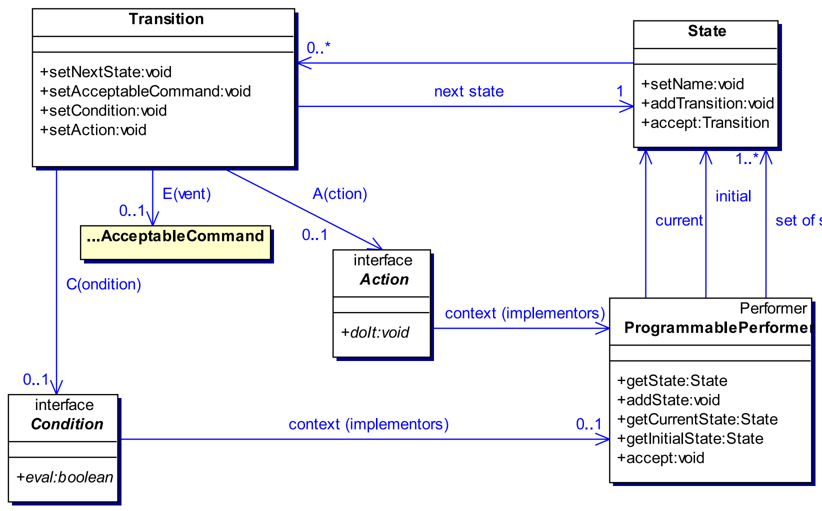


Figure 5.7: The `ProgrammablePerformer`

- `ProgrammablePerformer`, represents the state machine: it exposes a current state, the `accept()` method (to make it execute commands), and *the methods to manipulate the set of states* (i.e., `getState()`, `getStates()`, `addState()`, `getInitialState()`, `setInitialState()`);
- `State`, (set of, associated to the Performer) represents a particular *public* state of a Performer: it has a name and *the methods to manipulate the set of transitions* (i.e., `getTransition()`, `addTransition()`);
- `Transition`, (set of, associated to states) represents an 'arc' from one state to another, it is a 4-uple (AcceptableCommand, Condition, State, Action);
- `AcceptableCommand`, represents a command that may trigger a transition. The `match()` method evaluates as true only if the command given is acceptable (the acceptation policy depends strictly on the method implementation);
- `Condition`, represents an evaluable (boolean) condition. It may be a function of internal attributes or Visible values of the Performer;
- `Action`, represents an executable (method) action. It will be executed only after transition triggering.

It is very easy to create a new `ProgrammablePerformer` for a specific context. The programmer needs only to extend the `ProgrammablePerformer` class and write a specific constructor that defines an initial behaviour.

5.3.4 The `projections` package

In RTP framework `Projector` class reifies Projection. It is a special kind of `Performer` whose activity consists in aligning Performer visible variables. More in detail, a `Projector` instance aligns one imported visible to one or more exported visible.

Note that an arbitrary number of Projectors can be associated to the same visible exported by the Performer source. In addition, more than one Projector can exist even between the same pair of Visible exported by two Performers. This accounts for all those cases in which different strategies need to be applied when aligning the same information, or when (e.g, for fault tolerance) different underlying technologies coexist between the same pair of visibles.

Since the Projector role is to perform data distribution, the inherited method `accept(Command aCommand)` recognises only one kind of `Command` instance: the ‘sync’ one. At this aim, the `AcceptableCommand` class (see the `commands` package) is exploited. Once the command is accepted, the `synch()` method is performed.

Framework aim is not to cover implementation-related issues such as the communication technology and underlying platform (if any). The aim is to provide a general mechanism from which may kinds of Projectors may be implemented. For testing purpose, we have implemented a concrete Projector (whose coding is sketched in code fragment of figure 5.8). In this simple implementation, the `sync()` command aligns (copies) the values of source and target.

Other kind of Projectors may be realised, examples are Fifo buffers (see figure 5.9) in which either imported or exported visible variables are buffers. Obviously more complex situations can be devised and the `sync()` method is defined accordingly.

5.3.5 The `topologist` package

Topologist is special kind of Performer: its role is to create topology. At this aim the actions that it should be able to perform are the creations of Performers, Projectors linking Performers, and, eventually, the destructions of both Performers and Projectors.

The abstract class `Topologist` (sketched in figure 5.10) provides all the necessary methods to manage system topology. This class is a specialisation of `Performer` class and, consequently, it must be triggered to modify the topology with a suitable set of commands (i.e., `AccettableCommands` for this Performer).

```
/**
 * A Performer accepts commands (automaton-like) and allows access to
 * visibles by name. The role of this level is Visible management only.
 */
public abstract class Performer implements Nameable {
    /** The imported Visibles */
    private Hashtable in;
    /** The exported Visibles */
    private Hashtable out;

    /** Adds an imported Visible */
    public void addVisibleIn(Visible v, VisibleName n) {
        in.put(n, v);
    }
    /** Every Performer gives a name to every Visible */
    public Visible getVisibleIn(VisibleName name) {
        return (Visible)in.get(name);
    }
    //... same methods for exported Visibles

    /** Classes must be specialised to implement the appropriate accept policy */
    public abstract void accept(Command c);

    //...
}
```

Figure 5.8: The sync method of ConcreteProjector class

5.4 Dynamics

Even if the aim of RTP framework is to support the concrete design of systems dealing with real-time constraints, the framework may be exploited also for other kind of systems in which time does not play a crucial role.

In fact, despite timing issues, the RTP goal is to allow the dynamic planning of system behaviour at the programming level.

What this means is that the system behaviour may be “programmed” each time by observing what has happened.

The packages dealing with the controllable behaviour of the system will be presented in the sequel. Since classes dealing with system behaviour are organised into packages that do not take into account the typology of system they will support (with timing constraints or not), they are presented as they should be used.

First we will present classes useful to control system behaviour of deterministic system, then classes to use when building systems with temporal constraints.

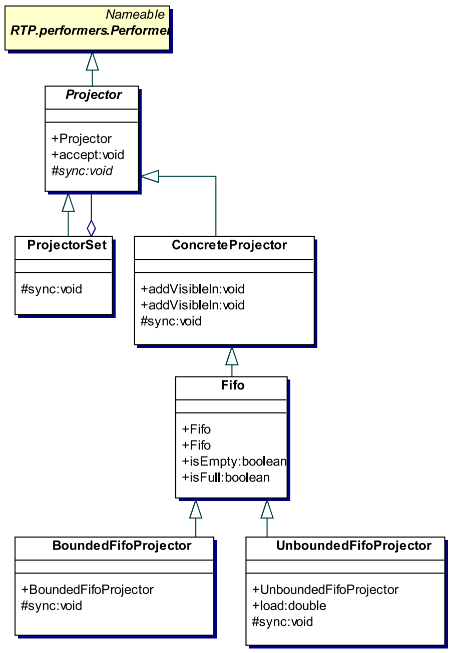


Figure 5.9: The `projections` package

5.4.1 System not dealing with timing constraints

Important concepts here are Traces and Requests that completely describe system behaviour.

As introduced in section 4.5.1, the actions performed by a system are arranged inside Trace. A Trance is a partially ordered set of Requests. RTP framework provides homonymous classes for both Trace and Request as shown in figure 5.11.

A pair defines the `Request` class:

- *recipient*, a Name class instance (implementing the Nameable interface) specifying the univocal recipient for the Request. The recipient may be any kind of Performer (a Performer, a Projector, or a Topologist);

- *command*, an instance of the Command class defining what the recipient has to perform.

`Trace` class is an aggregation of `Request` instances. To emphasis the separation between what has already happened and what has planned to happen, the dependency of `Trace` class with respect to `Request` class is realised using two aggregation namely *todo* (specifying what has planned), and *done* (specifying what has already happened). The aggregation labelled

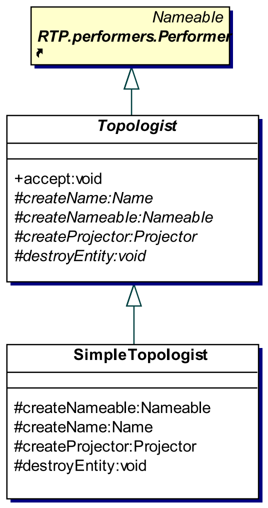


Figure 5.10: The `topologist` package

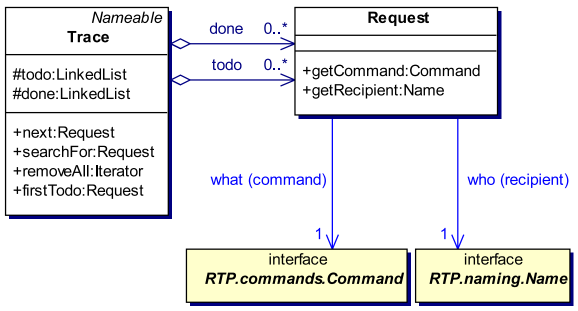


Figure 5.11: The `traces` package

todo reifies the FutureTrace, whereas the aggregation labelled *done* reifies the PastTrace (as desctribed in subsection 4.5.1).

The `next()` method constitutes the core of `Trace` class definition: it returns the next performable `Request` instance. When the selected Request has been delivered and executed, then it is moved in the PastTrace. The implementation of the `next()` method is strictly domain dependant. A simple implementation is provided in the `Trace` class: `Request` instances are stored using a First-In-First-Out (FIFO) policy, accordingly, the method returns the first Request in the pile. Obviously, this policy may be changed by subclassing `Trace` class and overriding the `next()` method as needed.

`Trace` class is passive, i.e., it is unaware about when its `next()` method is invoked. In general, the entity that is in charge of activating a Trace is the Engine. An Engine simply executes the method `next()` of all the Traces it controls. Then, on the basis of the `Request` instance returned by the `next()` method, it searches for the correct Performer by means of the NameServer, and, finally, it dispatches the command. The execution is sketched in the sequence diagram of figure 5.12.

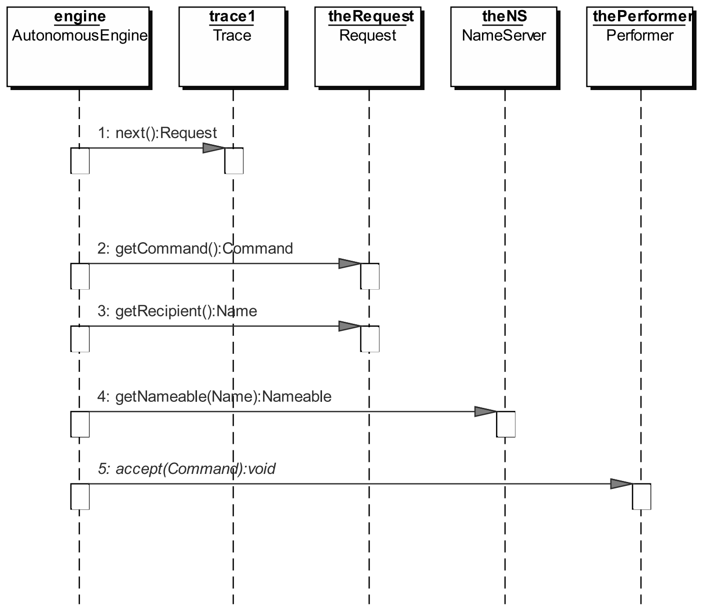


Figure 5.12: The execution of a Request

RTP framework provides a general `Engine` class. Its method `fetchAndDispatch()` is responsible for fetching the Request and correctly (in the sense of the correct recipient) dispatching it. This method is abstract and, consequently, the subclasses has to provide it a suitable implementation. In RTP are defined two concrete implementation of the Engine: `AutonomoutEngine` and `TickedEngine`. The former is used when there is no entity that is in charge of activating the Engine, the latter is used when someone else triggers opportunely the Engine (see section 5.4.2).

`AutonomousEngine` is a thread the execution of which may be controlled by means of its exported methods. When activated (method `activate()`), it continuously invokes the `next()` method of the Trace instances it controls.

The code fragment in figure 5.13 shows the construction of a Trace (lines 1, 2, 3, and 4) by adding specific Requests. Then, an Engine is instantiated (line 5) specifying which Trace it controls. Finally the Engine is activate (line 6). Since the Trace we used in the example is a `Trace` instance, then each time the engine invokes its `next()` method, the Request in the top of the FIFO is returned.

```
//...
Trace t = new Trace();                                  //1

t.add(new Request(
    new GenericName("Perf1"), new GenericCommand("Do"))); //2
t.add(new Request(
    new GenericName("Proj1"), new GenericCommand("SYNC"))); //3
t.add(new Request(
    new GenericName("Perf2"), new GenericCommand("Do"))); //4

Engine e = new AutonomousEngine(t);                     //5

e.activate();                                           //6
//...
```

Figure 5.13: Trace construction

5.4.2 Systems dealing with timing constraints

More interesting is the management of timing Requests in RTP framework.

Every concept relative to time is modelled inside the `time` package presented in figure 5.14.

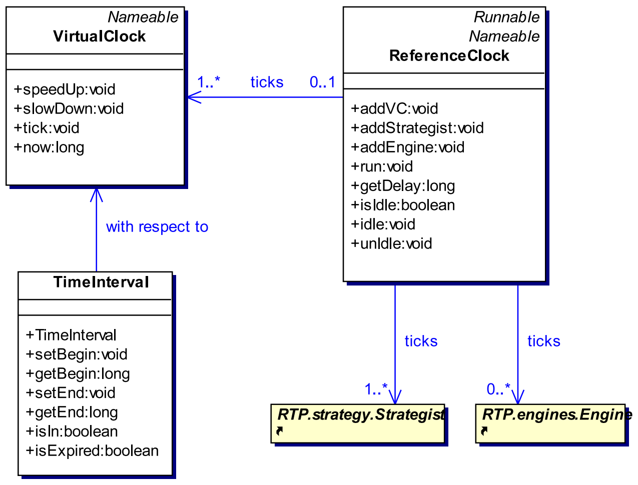


Figure 5.14: RTP classes reifying time

Here we have defined the following key classes:

- `ReferenceClock`, reifies the absolute time. It is the reference timer for a bunch of virtual clocks. It is responsible for advancing the real time;

- `VirtualClock`, models a virtual clock as defined in subsection 4.5.2. It is characterised by a starting point, an end point, and a period. The period is the number of reference clock ticks between updates of the virtual clock current time. `VirtualClock` is a “time-event” generator for entities interested in current time.

- `TimeInterval`, specifies the temporal coordinates with respect to a `VirtualClock`. A `TimeInterval` can be an instant ($begin = end$) or an interval.

Obviously, `Request` and `Trace` classes as defined in 5.4.1 must be specialised in order to manage timing issues. At this aim, the framework provides the `TimedRequest` and `TimedTrace` classes as skectched in figure 5.15.

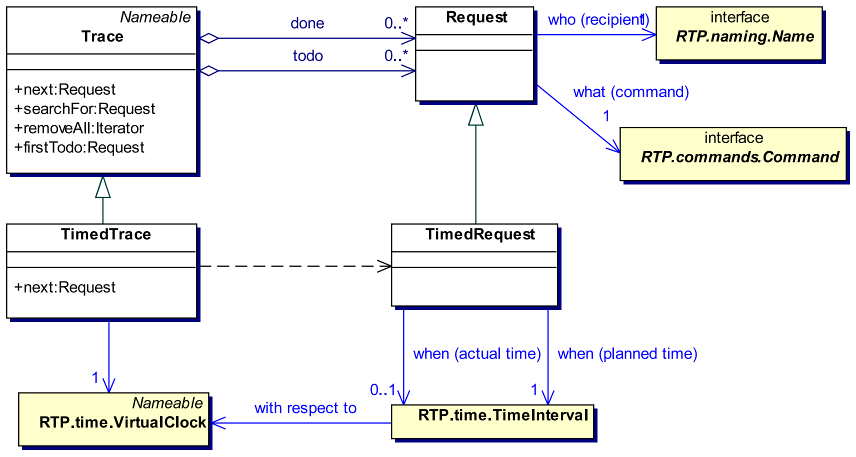


Figure 5.15: `TimedRequest` and `TimedTrace`

`TimedRequest` class is subclass of the Request one. It inherits the association with the recipient and the command to be performed. What is more in its definition is the association with at least one instance of `TimeInterval` describing the interval validity (in a temporal sense) for the Request. This association describes the *planned* temporal interval in which the Request have to be delivered and performed. The other association (labelled *when (actual time)*) within the `TimeInterval` class describes the *actual* (if any) temporal interval. This association describes the time in which the Request has been delivered and then executed. The cardinality of the association is 0..1 because it may occur that a Request cannot be executed since its planned time is expired with respect to the now value.

`TimedTrace` is a subclass of the `Trace` class. It contains `TimedRequests`, and, consequently, overrides the `next()` method with a time-aware implementation.

A `TimedTrace` is aware of the current time by means of the associated `VirtualClock`. When its `next()` method is invoked, it finds for the `TimedRequest` whose planned `TimeInterval` is near to current time.

As for `Trace` class, an Engine is in charge of activating the `TimedTrace` `next()` method. But, on the contrary, the Engine for timing system must be activated in a suitable way, i.e., it cannot be an autonomous thread[4]. RTP defines `TickedEngine` class as a class depending to the reference clock: it is "ticked" each time the reference real time advances. The reason why we do not associate the `TickedEngine` class to a `VirtualClock` is that an engine may "controls" several virtual clocks whose periods are different.

The sequence diagram sketched in figure 5.16 illustrates how classes interacts each other. In the same figure an object called Strategist, creates the `TimedTrace`, the `VirtualClock`, the `TimedEngine`, and the `ReferenceClock` instances, adjusting their relationships. When the reference clock is activated, it communicates the advancement of time both to the virtual clock (`tick()` method) and the Engine (`activate()` method). In turn, the Engine verifies if there are Requests to be delivered by the invocation of the `next()` method of the TimedTrace it controls. If a TimedRequest is performable, then it operates in the same way as described in section 5.4.1: Engine retrieves, by means of the `NamingServer`, the recipient Performer and invokes its `accept()` method by passing the command to be performed. At the end of the Performer processing activity, the actual `TimeInterval` is imposed to the `TimedRequest`.

5.5 Strategy

Strategy is in charge of defining the system behaviour. At this aim, it defines `Trace`s and `TimedTrace`s by the observation of the current state (`Visible` or `NotifyingVisible` exported by the Performer) and the past system behaviour (PastTrace in `Trace`s classes).

Even if Strategy encapsulates any domain-related issues, the RTP `Strategist` class represents a very simple reification of strategy. The aim of this class is to emphasises the role of the strategies inside a system. This class exports the following methods:

- `addTrace()`, adds a `Trace` or a `TimedTrace` to this Strategist. The added Trace will be manipulated by this Strategist only;
- `observe()`, allows the Strategist to control the system behaviour. This method may be invoked by the reference clock each time it ticks the Engine and the virtual clocks. The method is declared abstract since its implementation is strongly domain-dependant.

[4]Really, it can be an autonomous thread, but it will "tick" the `TimedTrace` keeping no account about the time

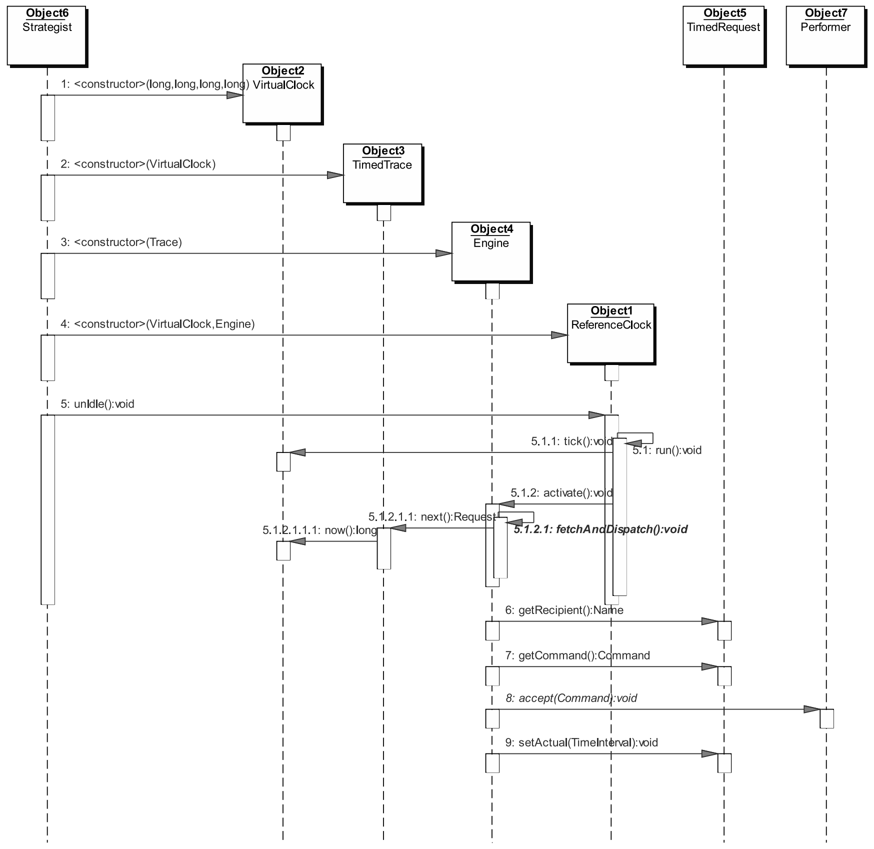


Figure 5.16: The sequence diagram for a simple timing system

Special kind of `Strategist` is the `NotifiableStrategist`: it is an observer of `NotifyingVisible`s instances (see subsection 5.3.3). The class export the method `update()` that is invoked when a `NotifyingVisible` instance notifies a change of its value.

`observe()` and `update()` methods are very useful when an event-drive policy is preferred. This is not the only way of controlling system behaviour: it is possible, also, adopting a polling policy.

Obviously, these are very simple classes, that the end user may or not decide to use. The classes have been introduced inside a specific package (`strategy` package) of the RTP framework only to emphasise the role of Strategy inside the system.

Strategist (by notification or by polling), in order to plan (or modify) future behaviour, must have visibility over:

- *current system state*;

- *past system behaviour.*

Current system state may be acquired by the Strategist accessing directly to the visible variables exported by the Performers. Past system behaviour may be accessed by the Strategist using methods exported by `Trace` and `TimedTrace` classes. Some examples:

- `lastDone()`, returns the last performed Request;
- `getDone()`, returns the list of delivered Requests;
- `getMissed()`, returns the Requests that have not been executed;
- `getActualDurationLastRequest()`, returns the actual duration of the last Timed Request;
- `diffPlannedActualEnd()`, returns the gap (in time) between the planned and the actual time of a specific Timed Request;
- ...

Exploiting the above methods, the Strategist may plan future system behaviour by executing one (or both) of the following:

- modify the virtual clock speed (e.g., when the difference in time between planned and actual ends (or begins) of Requests are too distance). A speed modification is made by means of the `speedUp()` and `slowDown()` methods exported by `VirtualClock` class. When a virtual clock is slowed down, then its period is enlarged. Symmetrically, when a virtual clock is speeded up, its period is reduced. Note that the change of the virtual clock speed allows to control the the execution speed of the activity whose Requests are placed on the TimedTrace controlled by the virtual clock.
- modify the Trace (e.g., modify the actual begin of a Request to make it nearer to actual time, add a new set of Requests if the planned are still all executed, and so on).

Finally, with respect to the abstractions identified in subsection 4.1.1, we have the mapping with concrete entities in RTP framework as sketched in table 5.1.

5.5.1 An example

[5] Since the management of temporal behaviour at the programming level is the core of the functionality provided by the RTP architecture, a simplified

[5]The content of this section was elaborated with Sergio Ruocco and Andrea Trentini.

Temporal Abstractions	**RTP concrete classes**
Action	Command
TimeInterval	TimeInterval
RealTimeLine	TimedTrace
Action in RealTimeLine	TimedRequest
A (programmable) Clock	VirtualClock

Table 5.1: Mapping between temporal abstractions and concrete reifications

example of an actual implementation will be provided. The description in the sequel is a portion of an article that has been submitted for acceptance in SAC 2004 conference.

The application domain is a adaptive MPEG Video player. A video player has been chosen as a reference problem for its own set of diverse non-functional requirements related with time: reactivity, adaptability, timeliness, and so on. Both problem and solution are simplified, as their aim is to give a flavour of the proposed approach. Four basic *functional* requirements must be fulfilled to play an MPEG video:

- *F1*, read a byte stream including the encoded data;
- *F2*, parse the data and reconstruct YUV[6] images;
- *F3*, convert YUV frames in RGB;
- *F4*, display RGB frames in the proper sequence.

Such basic requirements must be enriched by *non-functional* requirements mainly dealing with QoS issues:

- *NF1*, read promptly incoming data, or may be lost;
- *NF2*, adapt to fluctuating incoming data rate;
- *NF3*, absorb the variable duration of algorithms;
- *NF4*, display the frames with the proper timing.

The main goal of the system is to play the video correctly. Thus the NF4 requirement, which directly arises from user needs, is the main constraint that must be fulfilled. However, all of them must be considered to get a satisfactory solution.

The ability of the system to fulfill NF4 hinges both on the characteristics of the execution platform, which include, but are not limited to, its

[6]YUV is a colour space that splits colour information in one luminance channel (Y) and two chrominance channels, U and V.

performance, and on the intrinsic nature of the encoded data. It is apparent that the system is expected to execute the algorithms that parse raw data, reconstruct images, and convert them in a format suitable for display "fast enough" (NF3). But also be reactive and read promptly incoming data before they are lost (NF1). Furthermore, it must keep the raw data in a buffer that is large enough to cope with the expected variance in data, to not starve the parse and decode activities rate (NF2). Therefore, while NF4 can be considered an externally controlled constraint, NF1–NF3 express the correctness of the system temporal behaviour in terms that depend more or less directly on the intrinsic characteristics, or qualities of both the general abstract architecture and the specific implementation of the system as they emerge during execution. In particular, NF1 mirrors the behaviour of a realistic low-level data reading mechanism.

The example supposes that an underlying hardware device physically reads data from the input peripheral directly into main memory (DMA-like). Then it manifests data availability with a signal and the amount of data read through a variable. Upon signal receiving, the application should perform a 'logical read' operation before incoming data are lost, i.e., advances the current buffer pointer to the next available buffer. Thus, the duration of the 'logical read' operation can be reasonably considered bounded and not dependent on the amount of read data.

5.5.1.1 The RTP solution

The implementation was done by carefully wrapping an already existing free MPEG player [2] written in Java in terms of RTP entities. The wrapping operation involved code cleaning and the removal of embedded synchronisation policies, i.e. the architecture did not need any structural modification. Figure 5.17 shows the performers defined (by specialization of the `Performer` class) to build the video player:

- `Parser` (includes `Reader`): this is a pre-emptable thread that continuously reads and parses data bytes from the MPEG file (the rationale for this peculiar implementation choices is discussed below);
- `Converter`: accepts a `CONVERT` command, converts images from YUV to RGB;
- `Displayer`: shows converted images, accepts a `DISPLAY` command.

Even though the intrinsic structure of the software used justifies per se the merging of Read and Parse and their implementation with a thread, a brief discussion provides a useful rationale for more general cases. Since this implementation bases on the standard Java platform, not on a bare embedded system, it is reasonable to rely on the language run-time support for a suitable implementation of the Read Action from the input stream, and on the underlying operating system for the prompt read and buffering of incoming data from the device. With respect to the parsing operation, it has not definite temporal requirements per se except for “adequate performance”.

At the architectural level, the Parse Action has been modelled by an atomic Action with a definite start, an unpredictable end, and an infinite deadline.

On the other hand, facing the task of implementing an all-software system that in principle must be suitable for a single-CPU platform, to realise the Parse Action exactly as previously conceived would likely interfere with the execution of others time-critical Actions. In this case a standard, preemptable thread seems to be the most natural solution. In this implementation the Parser is a wrapper that hides the peculiar implementation of the reading and parsing activities. When the asynchronous parsing thread completes an image, the Strategist notes it as soon as a Parse Action is “performed”.

Of course, like other threads in the system, the Read/Parse thread is always preempted by time-critical Actions on the real-time line.

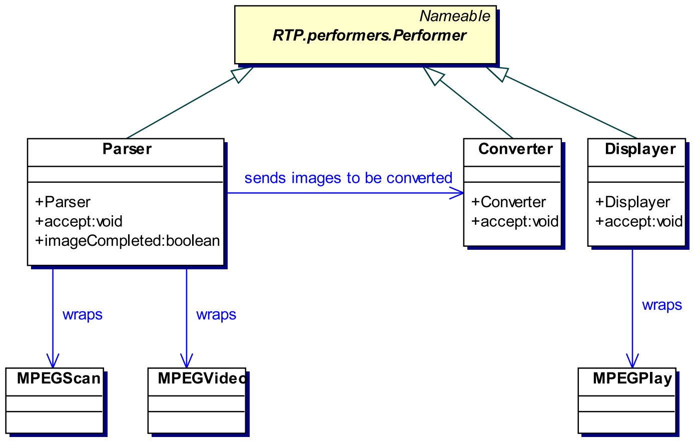


Figure 5.17: Performers in MPEG Video Player application

Part of the Java implementation is shown in code fragments presented in figures 5.18, 5.19, 5.20, and 5.21.

Code fragment in figure 5.18 belongs to the bootstrap code: creation of a `TimedTrace` with its associated `VirtualClock` (comments 1 and 2) and an `AutonomousEngine` (comment 3). At the end of the code, when the Engine is already set up and running, control is given to the `Strategist` by calling its `run()` method (comment 4).

Code fragment in figure 5.19 shows the `Strategist` behaviour. When the `AutonomousEngine` is running, i.e., continuously fetching and dispatching Requests, the `Strategist` role is to observe and control the overall

```
//...
/** Creates a virtual clock */
VirtualClock vc = new VirtualClock();          //1

/** Create an empty trace */
TimedTrace tr = new TimedTrace(vc);            //2

/** Creates engine and set trace to be fetched */
AutonomousEngine eng = new AutonomousEngine(); //3
eng.addTrace(tr);

/** Gives control to the strategist */
Strategist.run();                              //4
//...
```

Figure 5.18: Creation of an empty trace (with a virtual clock)

```
/** The run method for strategy control */
public static void run() {
    initialStrategy();        //1
    while (true) {
        strategy();           //2
    }
}
```

Figure 5.19: The Strategy

system state. The `Strategist` must build the initial trace (comment 1) and periodically adjusts trace content (comment 2).

Code fragment in 5.20 represents a possible startup system strategy. Since NF4 is the primary temporal requirement, the Strategist can fulfill it by planning, once and for all, a sequence of display actions.

Finally, code fragment in 5.20 sketches the dynamic behaviour of the system. When the state changes, the Strategist plans future behaviour as follows:

- if the performed action was a Parse and the image is completed (comment 1), then the Strategist plans a Convert (comment 2);

- if the action performed was a Read (comment 5), the Strategist may modify the previous planned corresponding Parse action to change its `plannedBegin` constraint. In other words, the Strategist tries to plan the Parse action at a time that is earlier than the previously planned

```
public static void initialStrategy() {
    /** Plan maxFrameNumber frames display in advance */
    for (int i = 0; i < maxFrameNumber; i++) {
        TimeInterval tInt = new TimeInterval(
        i*frameInterval, i*frameInterval+displayDuration);
        Main.tr.addAction(new TimedRequest(tInt, "DISPLAY", "Displayer"));
    }
}
```

Figure 5.20: Initial strategy

one. It can do it by retrieving in the past timeline the `actualEnd` of the Read action and assigning it to the `plannedBegin` of the corresponding Parse action (comments 6, 7, and 8). Even if somewhat redundant, this specific planning emphasises how the observation of the past behaviour may change the future behaviour;

- if the reading mechanism flags that new data are available (comment 9), the Strategist plans a pair of Read and Parse actions (comments 10 and 11, comments 12 and 13 respectively).

Experimental results have shown that the proposed methodology does not affect neither performance nor functionality since the RTP player is indistinguishable from the original version.

```
public static void strategy() {
    if (Main.tr.last().equals("PARSE")) {
        if(ns.getNameable("Parser").imageCompleted()) {           //1
            /** Plans convert, a YUV image is ready */
            TimeInterval tInt = new TimeInterval(Main.tr.now(),   //2
            Main.tr.next("DISPLAY").getPlanned().getBegin());     //3
            Main.tr.addAction(new TimedRequest(                   //4
                tInt, "CONVERT", "Converter"));
        }
    }
    if (Main.tr.last().equals("READ")) {                          //5
        /** Modifies the previously planned parse */
        TimeInterval tlr = Main.tr.last().getActual();            //6
        TimeInterval tlp = Main.tr.next("PARSE").getPlanned();    //7
        tlp.setBegin(tlr.getEnd());                               //8
    }
    if (ns.getNameable("Reader").available()>0) {                 //9
        /** Plans a read and a parse */
        TimeInterval readtInt = new TimeInterval(                 //10
            Main.vc.now(), Main.vc.now() + readDeadline);
        Main.tr.addRequest(new TimedRequest(                      //11
            readtInt, "READ", "Reader"));
        TimeInterval parsetInt = new TimeInterval(                //12
            Main.vc.now()+readDeadline, Main.vc.getEnd());
        Main.tr.addRequest(new TimedRequest(                      //13
            parsetInt, "PARSE", "Parser"));
    }
}
```

Figure 5.21: The strategy

Chapter 6

CLAM Approach Applied

6.1 Introduction

To verify the validity of the approach we propose to solve the “chicken and egg” problem (as described in chapter 3), we have translated CLAM idea into a concrete software implementation.

The design of the software system has been made according to Real-Time Performers architecture, whereas its implementation has been made exploiting the Real-Time Performers framework. The motivations of this choice are manifold, among them we report:

- the architectural separation of concerns between data computation, data distribution, and both data computation and data distribution activations;
- the management at the programming level of the system temporal behaviour which involves, among the others, the dynamic change of activities relative execution speed.

The separation of concerns allows to obtain reuse. This is essential to support any future development concerning the algorithms involved in each steps of *Perception*, *Localisation*, and *Modelling*. The management of the system temporal behaviour allow to face issues concerning the criticalities.

This aim of this chapter is to provide a description of the algorithmic aspects of our implementation. In detail, this chapter will provide a description of the methods we use to implement each step of the *Perception*, *Localisation*, and *Modelling* activities.

The description of both the system design and its concrete implementation will be provided in chapter 7. In the same chapter, details concerning the system dynamics will be faced.

To emphasise the generality of CLAM approach, we remark that what will be presented in the following sections is only one way of implementing the activities.

Whatever it will be the desired implementations, when designing a CLAM system, it is important to treat separately the activities in order to exploit CLAM principles.

6.2 Perception Activity

Perception activity addresses all the issues concerning the inputs captured by the robot sensorial equipments. The aim of this activity consists in acquiring data from the perceptive devices and elaborating them to produce suitable information needed by both the *Localisation* and *Modelling* activities.

Referring to chapter 3, *Perception* activity must provide for CLAM purpose the following kind of information:

- *perceived robot pose*: the robot pose (i.e., position and orientation) inside the environment;

- *perceived view*: a representation of the environment made up of geometrically located objects captured when the robot was at the corresponding perceived pose.

Perception activity generated *perceived located views*, i.e., pairs *perceived view, perceived pose.*

Perceived located views depend on the typology of the robot and the devices it uses. In our experiments, we use a Robuter endowed of:

- 24 sonar sensors able to perceive the presence of an obstacle in a range from 15 cm up to 600 cm, with an accuracy of the order of the centimeter; the cone of the sonars is around 25° wide (not used for our scope);

- a trinocular video system;

- an odometric system;

The trinocular video system is used to acquire images from which reconstructing the *perceived view*; whereas the odometric system is used to acquire the *robot pose.* At the end of the *Perception* activity the following information will be available:

- *perceived view*, reifying the CLAM perceived view PV. It is a set of 3D segments (geometrical located objects, as defined in CLAM) representing the robot visual perception about the scene. As the robot moves, it acquires a triplet of images, one from each camera. Those images are then elaborated to produce the robot current perceived view consisting in a set of 3D segments. The robot founds the reference frame to this set of segments.

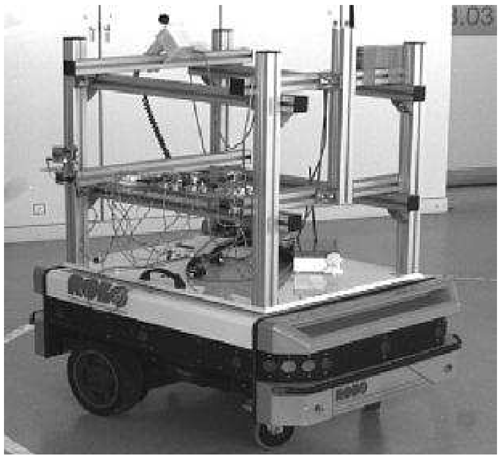


Figure 6.1: The Robuter™

- *perceived robot pose*, reifying the CLAM perceived pose PP. It describes the robot position and orientation as provided by its odometric system. This pose is referred to an initial reference frame.

Perception activity performs the following steps:

- *Perceived View Acquisition*, generating the *perceived view*;
- *Perceived Pose Acquisition*, generating the *perceived robot pose*.

Since the pose and the view are related information, these steps must be executed at the same time.

6.2.1 Perceived View Acquisition

The aim of this step is to provide the other activities with a suitable representation of the perceived view. In this specific implementation, the perceived view consists in a 3D representation (3D view) of the environment likewise perceived by the three cameras. A 3D view is constituted by exactly one set of 3D segments whose reference frame is relative to the robot.

This step needs the following phases to be executed (see figure 6.2) in the order in which are listed:

1. *Images Acquisition*: acquires one digitalised image from each camera;
2. *Filtering*: reduces the amount of noise affecting the images; preprocesses them to produce for one of each an array of pixels expressed in the image coordinate system;
3. *Edges Detection*: filters each images to obtain for each image a set of significant features (in our case, edge points);

4. *Polygonal Approximation*: determines, from each sets of edge points, the corresponding 2D segments;

5. *Stereo-matching*: retrieves triples of homologous 2D segments in the three image planes;

6. *Triangulation*: merge each triples of 2D homologous segments to obtain the 3D segment in space coordinate.

Even if 3D view reconstruction is out of the scope of this work, a brief explanation of the previous phases will be provided in the following subsections.

6.2.1.1 Images Acquisition

This phase drives image acquisitions from the three cameras. As the result of the acquisition process, three digitalised images, one from each camera, are available for further computation. Figure 6.3 shows an acquisition from each camera executed during a test phase. Those images will be processed by the "Filtering" phase.

Before entering into detail of the 3D view reconstruction, a brief explanation about the image formation will be provided.

The cameras are modelled with the *pinhole* model [48]. Each camera is modelled by its *optical center* C and its *image plane* P. A point $P(x, y, z)$ in the 3D space projects onto the image plane at an image point $I(u, v)$ (see figure 6.4).

This transformation (from P to I) is modelled by a linear transformation $\mathbf{T}$ in homogeneous coordinates (perspective transformation).

If $I^* = [U \quad V \quad S]^T$ are the homogeneous coordinates of point I and $[x \quad y \quad z]^T$ are the coordinates of a generic point P in the scene, the following relation holds:

$$I^* = \begin{bmatrix} U \\ V \\ S \end{bmatrix} = \mathbf{T}[x \quad y \quad z \quad 1]^T \tag{6.1}$$

where $\mathbf{T}$ is a 3×4 matrix termed *perspective matrix* of the camera also called Direct Linear Transformation (DLT) matrix. The determination of the twelve (eleven are independent) parameters of $\mathbf{T}$ is made adopting a experimental approach (*camera calibration*). The reason for this choice is that the effective transformation from the absolute reference frame to the camera reference frame depends on a number of physical parameters that are not known. Those parameters are termed *intrinsic* and *extrinsic* parameters. The camera calibration approach is the process of estimating those intrinsic and extrinsic parameters. The parameters to evaluate are eleven instead of twelve since we deal with homogeneous coordinates. Six of them are extrinsic and interest the camera (i.e., the reference frame of the camera with respect to the reference frame of the real world), the other five are

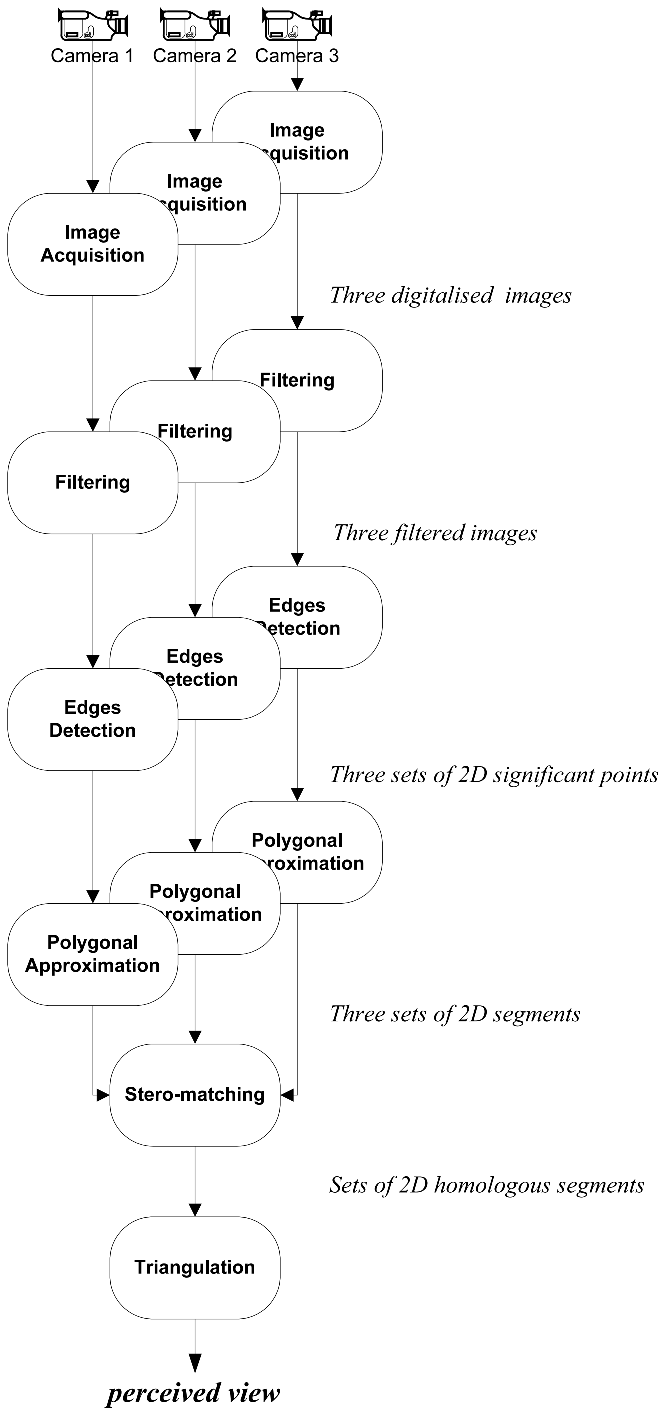


Figure 6.2: Perceived 3D view reconstruction steps

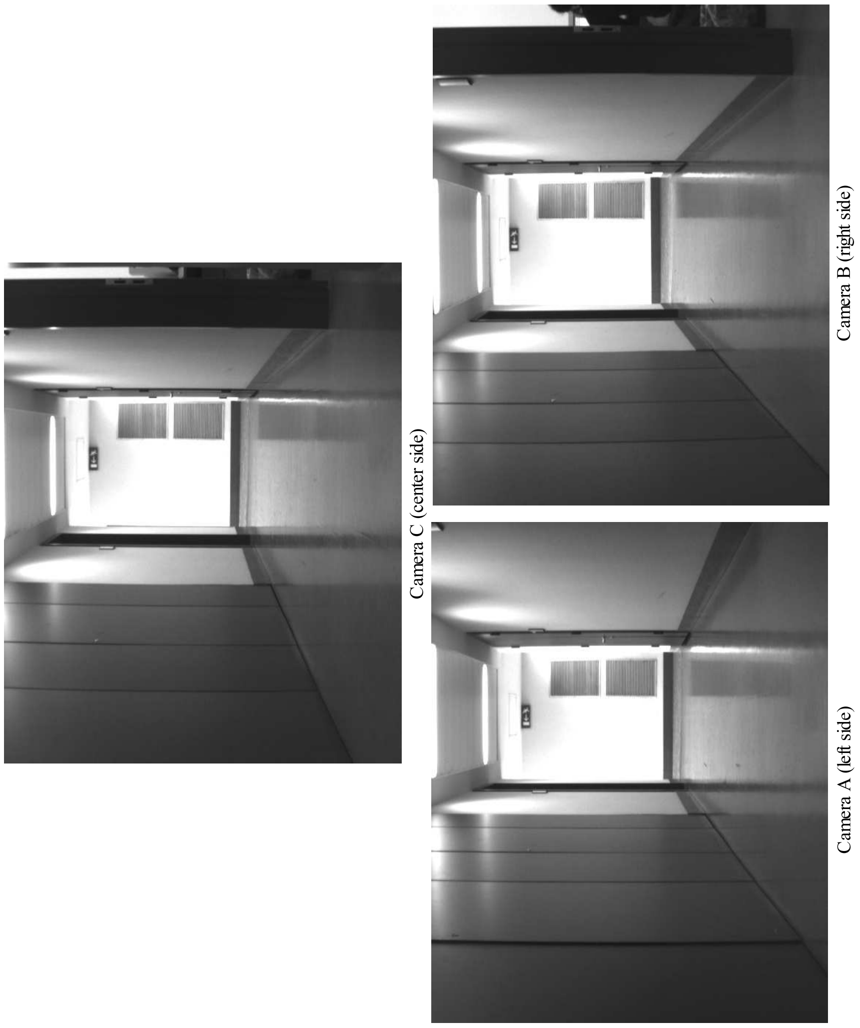


Figure 6.3: Three images acquired by cameras on Robuter™

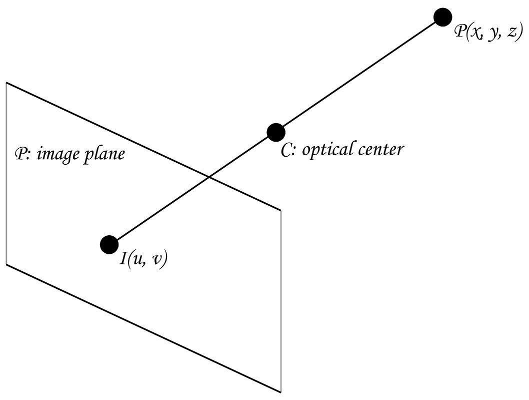


Figure 6.4: The pinhole camera model

intrinsic and are related to the reference frame of the image plane (three are referred to rotational and translational parameters, one to the scale factor, and the last concerns the position of the images axes u and v).

The estimation process assumes that both the 3D vectors $\mathbf{M_i}$ of N reference points M_i and the 2D image coordinates (u_i, v_i) are given, and consider the problem of estimating $\mathbf{T}$ from these measurements. Six non-coplanar points are sufficient [40]. In practise, dozens of points are used, and the parameters of $\mathbf{T}$ are computer either by least squares or by Kalman filtering.

6.2.1.2 Filtering, Edge Detection, and Polygonal Approximation

Filtering is the first phase in the image processing elaboration. The aim is to reduce the noise in each image: each image preprocessed to increase its signal noise ratio (SNR).

Then, Edge Detection phase discovers the contours of objects in a scene. It is an operation difficult for a computer. Some of the reasons are the following ones:

- the notion of object is not understood. Indeed, the goal of computer vision is to identify objects in scenes;
- an edge is a discontinuity of some sort of the image intensity function. We have the problem of measurement noise. This intensity is a physical measurement that is subject to noise. The detection of discontinuities of image intensity can be achieved mathematically by computing derivations of this function;
- the source of an edge may be of different type: some edges comes from shadow cast by objects, some from variations in the reflectance of objects, and so on.

This operation is generally performed analysing the image luminance intensity: qualitatively, image contours correspond to *discontinuities* of the luminance intensity. Contour points are defined in regions of transition corresponding to a strong luminance intensity gradient[1]. A lot of extraction techniques are based on this consideration (examples are zero crossing and local maxima, see [6] for details).

Next step in image processing elaboration is the Polygonal Approximation of the points extracted from Edge Detection. The application of a Polygonal Approximation replaces all quasi-linear portions of bounds with straight segments. The first step groups points in blobs using a similitude criteria (e.g., the gradient direction). Once obtained the blobs, they may be approximated to segments using techniques applied like recursive splitting, Berthord algorithm, and others (see [6]).

The technique used in our implementation is quite similar to Fast Line Finder [58] algorithm. The sequence of operations performed are the following:

- module (see figure 6.5) and direction (see figure 6.6) calculus of the luminance intensity gradient executed for every pixel in the image;
- quantization of the gradient direction for every pixel;
- pixels clustering to create blobs (see figure 6.7) containing points that are potentially constituting a segment. This operation is performed by grouping pixels that have the following common characteristics for the gradient: its module over a prefixed threshold and with an homogeneous direction;
- the estimation of the segment presents in every blob with least square (LM) algorithm.

The result of Polygonal Approximation is a set of 2D segments in the image plane for each camera.

6.2.1.3 Stereo-Matching

Stereo-matching phase deals with the problem of relating segments in the three image planes, i.e., finding triplets of homologous segments that are projections of the same segment in the 3D space. The result of this phase is a set containing triples of homologous 2D segments.

The techniques adopted in literature are divided into two principal categories: methods that use luminance schemas, and methods that execute matching of contours. The method adopted in CLAM belongs to the second categories and executes a research of the correspondence between segments.

[1]How much strong is one of the problems.

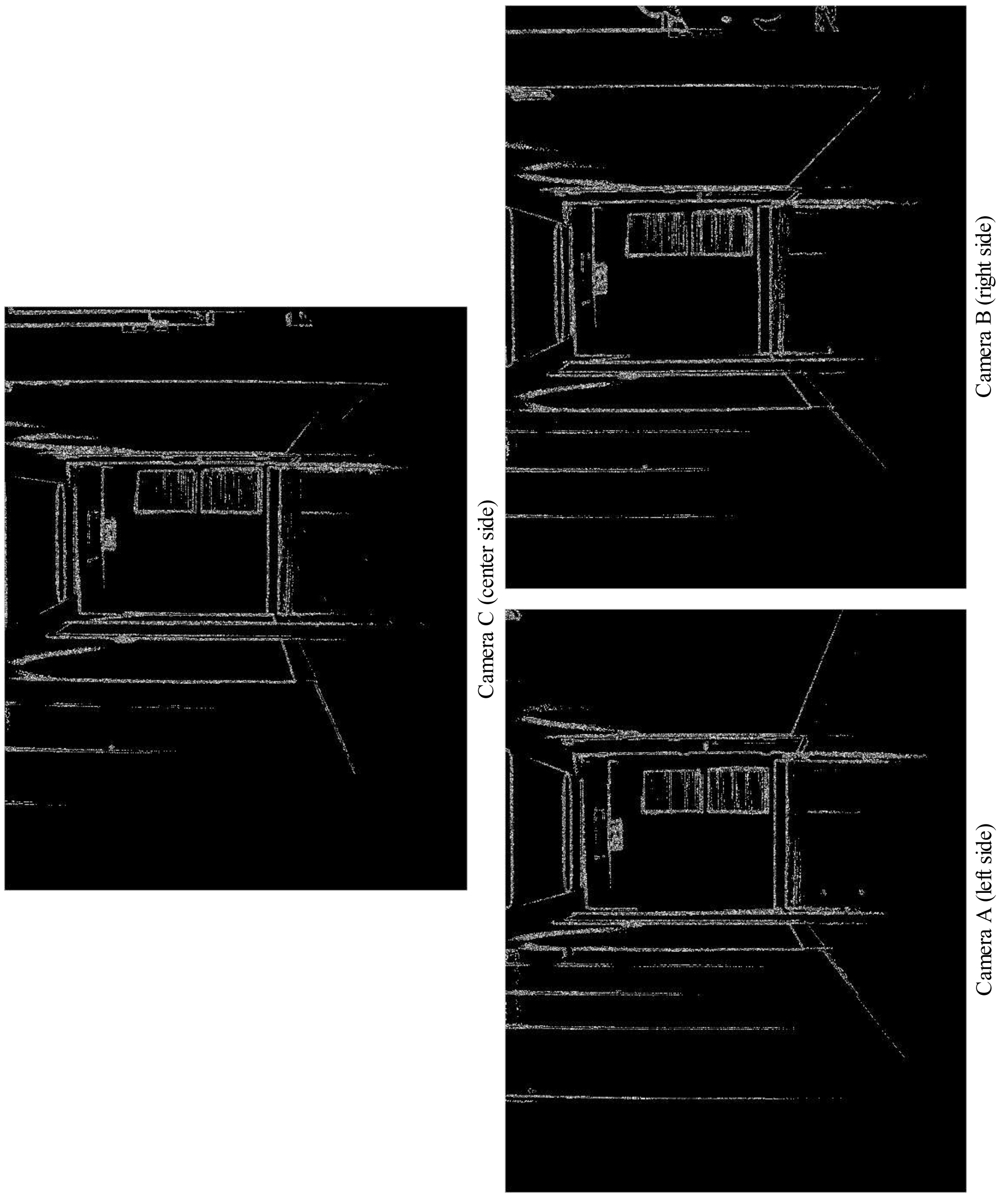


Figure 6.5: The gradient module

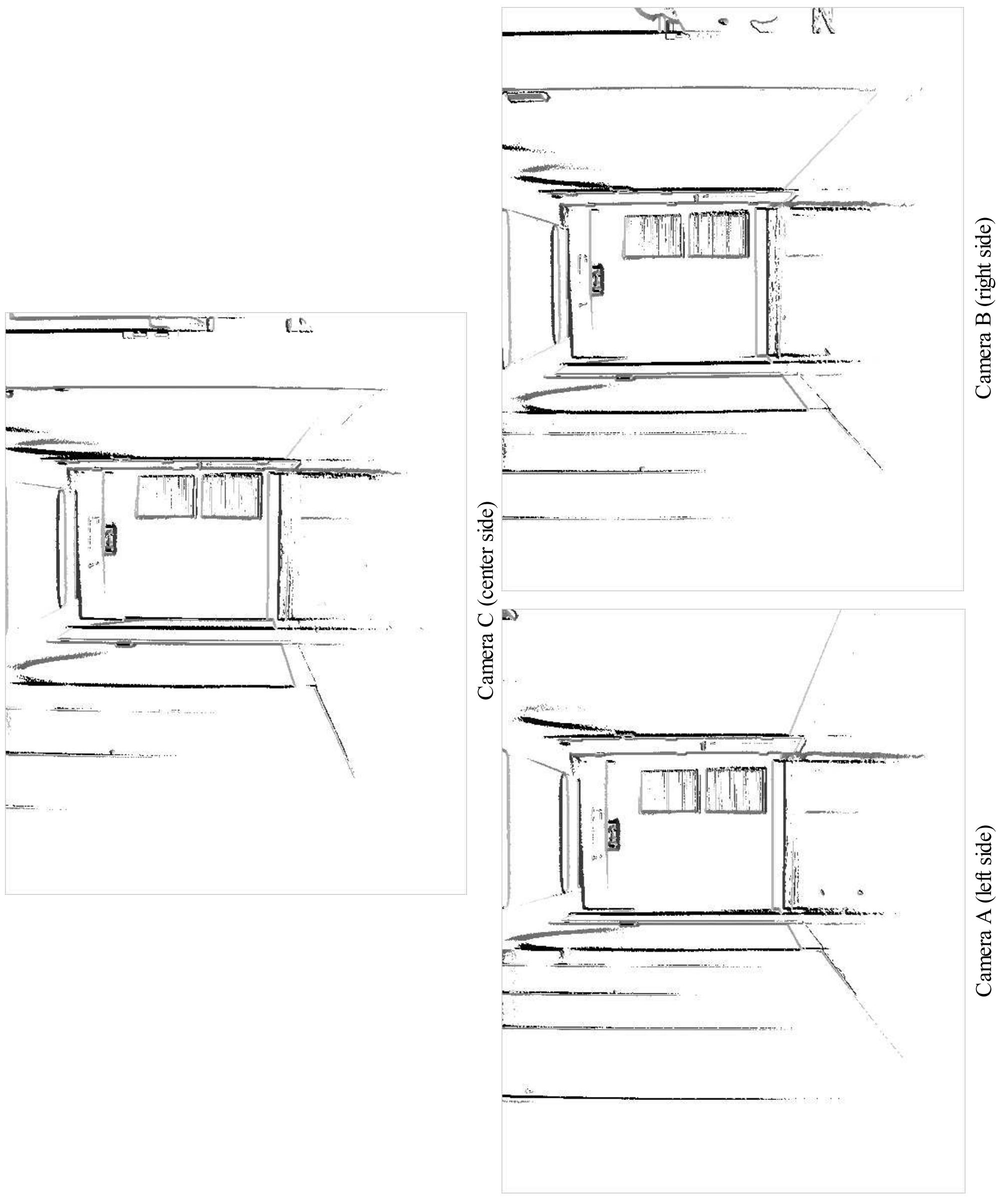


Figure 6.6: The gradient direction

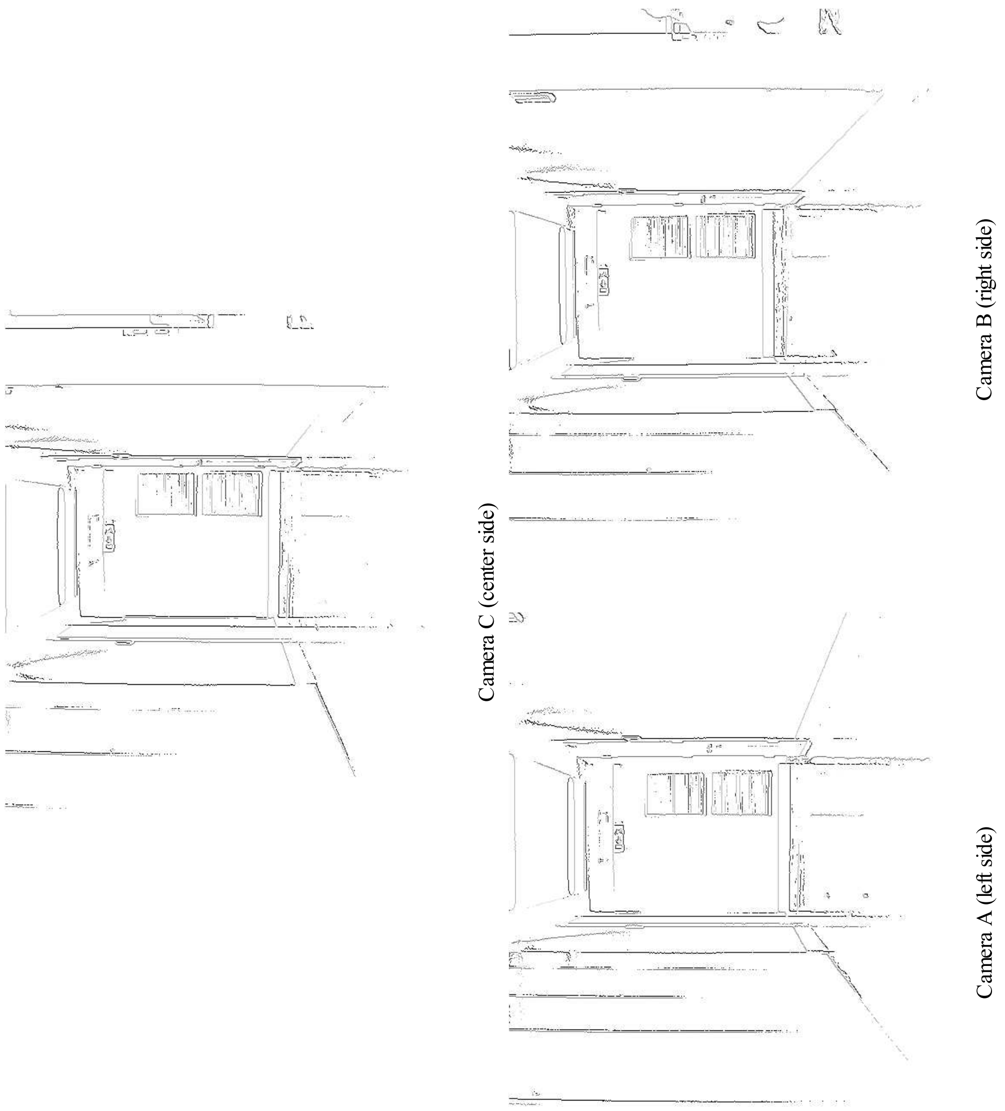


Figure 6.7: The blobs

A brief introduction to the stereoscopic vision and the related problem of matching will be provided, emphasising the necessity in introducing the third camera. For further information the reader can refer to [6] and [41].

We use more than a camera since it is not possible to deduce the geometry of an observed scene from a single image. Instead, it is possible to determinate the position of points observed if multiple images of the same scene are taken from different viewing angles. The process of combining multiple images of a scene to extract three-dimensional geometric information is called *stereo vision*. The simplest stereo process uses two images and is named *binocular* stereo vision. The term *trinocular* stereo vision is used when three camera are used.

Using a plurinocular system, we have the problem of relating the information of each camera. This problem is known as *stereo matching problem*. Given n camera, and n sets of planar features (points, segments, blobs, and so on) extracted from the images, how to detect homologous features, i.e., how to detect features belonging to different images that are projections of the same feature in the 3D space?

One solution to the matching problem resides in the geometry of stereo vision. We will describe the geometry of stereo vision in a binocular system, and we will show how a third camera will be useful to disambiguate matches.

Given point I_1 on image plane π_1, we will seek its homologous I_2 on the image plane π_2 (see figure 6.8). We define:

- *base line*, the line passing through the two optical centers C_1 and C_2;
- *epipolar plane*, the plane passing through the base line and the 3D point P;
- *epipolar lines*, the intersections between the epipolar plane and the image planes. Referring to figure 6.8, D_{21} is the epipolar line on the image plane π_2 associated with I_1, while D_{12} is the epipolar line associated with I_2. Those lines are called conjugate epipolar lines.

Given I_1, the point I_2 will belong to the epipolar line D_{21}. This is justified by the following considerations. The set of 3D points the image of which is the point I_1 belongs to the line connecting I_1 and C_1. The image of this line on camera 2 is the epipolar line associated with I_1 (conjugate epipolar line D_{21}). The problem is completely symmetric. If we consider the epipolar plane, this one intersects the images planes π_1 and π_2 along the conjugate epipolar lines D_{12} and D_{21}. Any point of the epipolar line D_{12} has its potential match in the conjugate epipolar line D_{21}, and vice versa. Any epipolar line of image 2 is the image of a line passing through C_1, and, consequently, the epipolar lines of image 2 form a bundle of lines with center in point E_2 (the image of C_1 in camera 2). Point E_2 is termed *epipole* of image 2. The same holds for camera 1: E_1 is the image of C_2 in camera 1 and so it is the epipole of image 1.

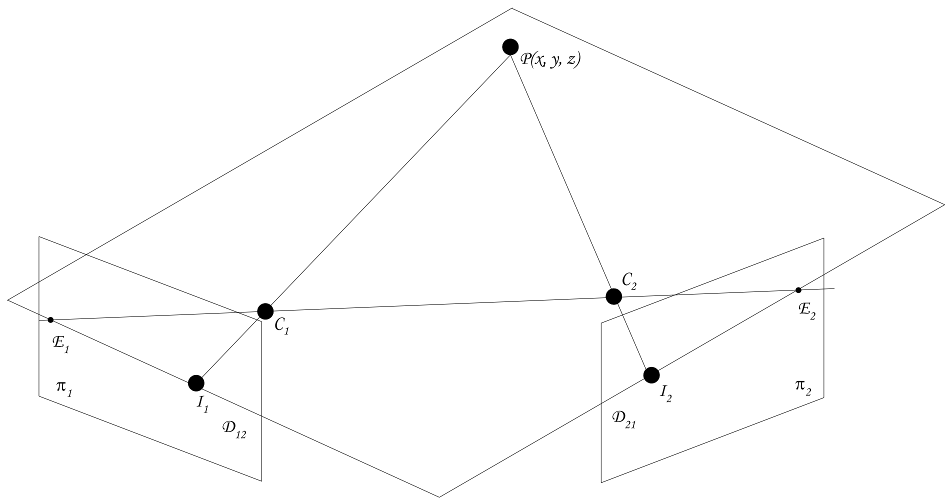


Figure 6.8: The geometry of a binocular stereo video system

Once the perspective transformations matrix of camera 1 and camera 2 are known, then the epipoles may be computed together with the epipolar lines (see [6] for details). The computation of the epipolar lines (that is called geometric constraints of stereo vision) is not sufficient to determinate stereoscopic matches. For a point in image 1 there exist an infinity of possible homologous points in image 2. Additional physical constraints are needed. Those are divided into local and global constraints. Considering segments to be matched, in the first set we find orientation and length, while in the second, uniqueness and ordering.

If we introduce a third camera (trinocular stereo vision) the validity of a match could be tested directly by verifying the presence of a specific point in image 3. This consideration leads to gain simplicity, reliability, and accuracy. For a comparatione between binocular and trinocular stereo vision see [26] and [25].

The geometry of a trinocular stereo video system is sketched in 6.9. Referring to that figure, we have:

- three cameras modelled by their optical center C_j and their image plane π_j, where $j = 1, 2, 3$;
- given a 3D point $P(x, y, z)$, its I_j image in camera j is given by the intersection of the line passing throw P and C_j with the image plane π_j;
- points I_1, I_2, and I_3 are homologous;
- every camera pair (i, j) satisfies the epipolar constraints. Thus, a point P in the real space will produce three epipolar planes whose in-

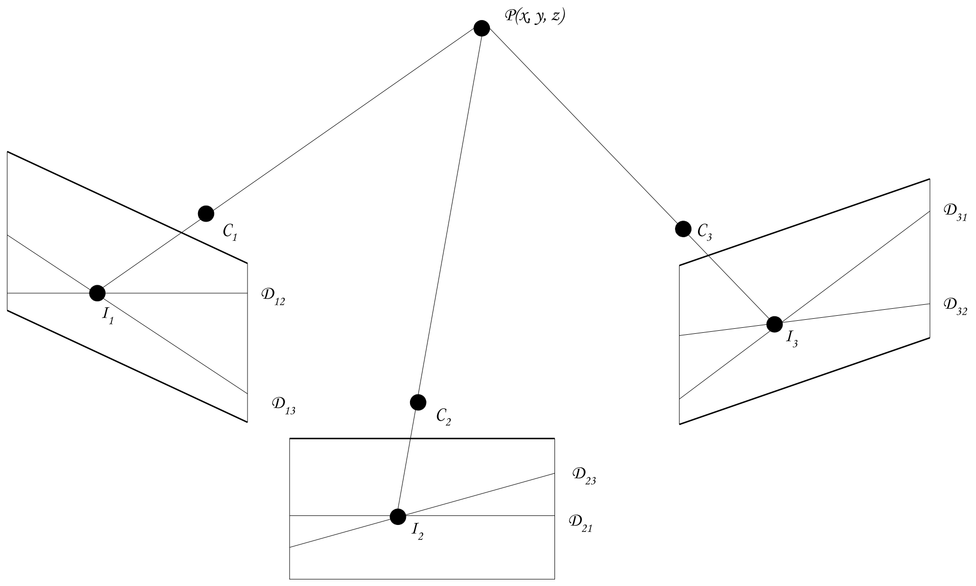


Figure 6.9: A Trinocular video system and the epipolar lines

tersections with the images planes will provide three pair of conjugate epipolar lines;

- if a triple I_1, I_2, and I_3 are homologous, every I_i will be at the intersection of the two epipolar lines $D_{i,j}$ and $D_{j,i}$ (associated to the other homologous I_j and I_k);
- to verify if a pair of image point I_i and I_j are homologous, it is sufficient to test the existence of the third homologous at the intersection of the epipolar lines $D_{k,i}$ and $D_{k,j}$ inthe π_k image plane.

The technique adopted in CLAM exploits both the trinocular epipolar constraints described above and length as local constraint.

The algorithm may be summarised as follows. Defining S_1, S_2, and S_3 the sets of segments respectively in image 1, 2, and 3, given a segment s_{1_i} in S_1, then the algorithm:

1. computes the epipolar lines of its extremes (A and B) in image 2;
2. finds in S_2, all the segments intersecting the two epipolar lines or that segments having extremes quite near to the epipolar lines. The result is a subset of S_2 termed S_{2_G} containing the sets of candidate segments for matching;
3. computes the epipolar lines of s_{1_i} extremes (A and B) in image 3;

4. for every segment in S_{2_G},
   (a) computes the epipolar lines of its extremes in image 3;
   (b) calculates the intersections with the epipolar lines evaluated at point 3;
5. finds in the set S_3 the segment that is more near and elects it as the virtual candidate segment;
6. finds near the virtual segment, that real segment that satisfy the local constraint (length).

The results of this phase are sets of homologous segments. Each set consists of one segment from each image. Figure 6.10 shows the results of the stereo-matching.

6.2.1.4 Triangulation

The aim of this phase is to provide the current perceived view, in our case a perceived 3D view constituted by 3D segments.

The previous phase (stereo matching) provides a set of homologous segments in the image planes. This phase deals with the identification of the three-dimensional segments that have produced the triplets of homologous segments. This operation searches the 3D segment that better approximates the three image segments that belong to a triplet. This operation consists in two steps:

1. identification of the *support line* R of the three-dimensional segment D;
2. identification inside R of the segment D.

Step 1 is made using the Kalman filter to obtain the optimum estimation of the parameters of line R. The process starts from the parameters of the two-dimensional lines that support the three homologous segments. The measurement equation is $f(x_i, a) = 0$. An estimation of the initial parameters is made using the complete representation of a line. Once obtained the initial estimate $\hat{a}_0$ and its covariance matrix S_0, next step is to linearized the equation of measurement $f(x_i, a) = 0$ whit values quit near to the current estimation $(\hat{a}_i, \hat{a}_{i-1})$. Figure 6.11 sketches the construction of the line R from the estimated segments d'_1, d'_2, and d'_3.

Step 2 deals with the detection of the 3D segment inside the line R. At this aim, the interpretation lines a_iC_i and b_iC_i are calculated for every image segment d_i where $i = 1, 2, 3$. Point A_i and B_i on line R are obtained such as the distance from lines a_iC_i and b_iC_i respectively is minimum. This operation permits to compute segments $\overline{A_1B_1}$, $\overline{A_2B_2}$, and $\overline{A_3B_3}$. Noise

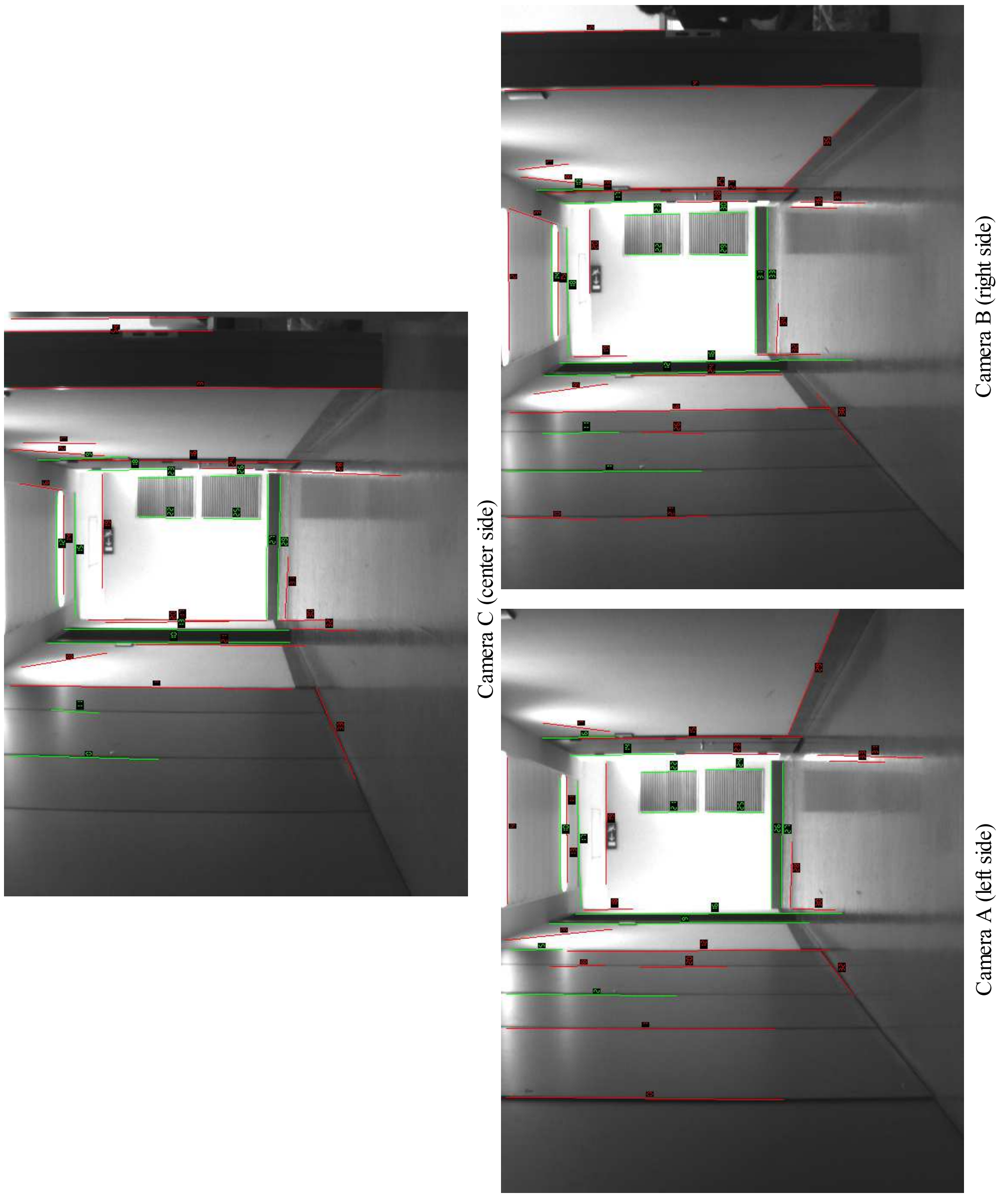


Figure 6.10: Stereo-matching results

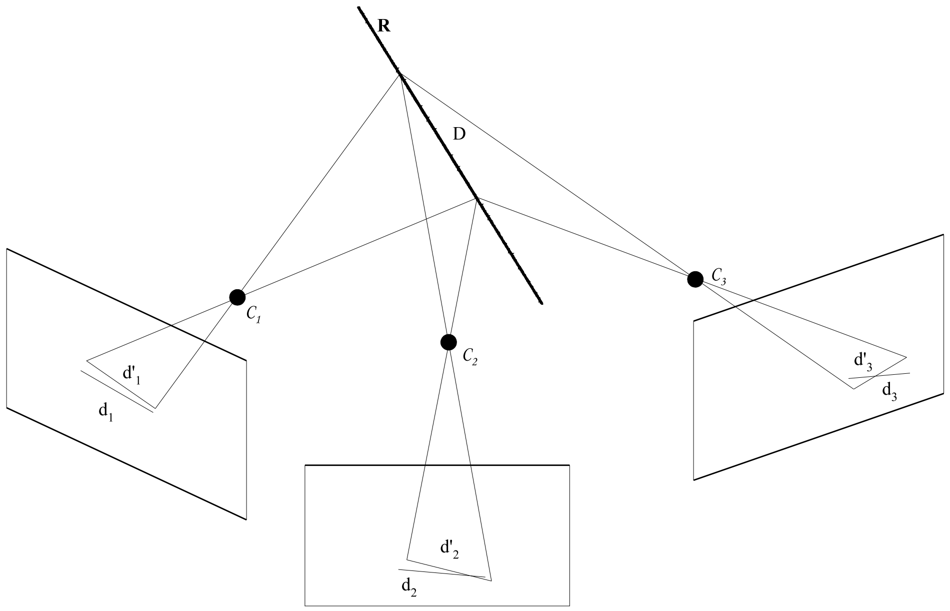


Figure 6.11: Construction of the line R

corrupting images, leads to have that $A_1 \neq A_2 \neq A_3$ and $B_1 \neq B_2 \neq B_3$. The segment D, image of the triplet d_1, d_2, and d_3, is so determined as intersection of segments $\overline{A_1B_1}$, $\overline{A_2B_2}$, and $\overline{A_3B_3}$.

Figure 6.12 shows the construction of the interpretation lines for segment d'_i and the identification of its extremes in the 3D space, whereas figure 6.13 shows the construction of a 3D segment as intersection of the three 3D segment that have produced the projections d'_1, d'_2, and d'_3.

At the end of the "*Perceived View Acquisition*" phase all the sensorial information coming from the three camera are *fused* to generate a set of 3D segments representing the perceived view and whose reference frame is referred to the robot, i.e., the robot kinematics center is the origin of their reference frame.

Finally, the segments returned by the *Perception activity* are enriched with a factor of uncertainty. The uncertainty is represented by a 6×6 symmetric matrix expressing the covariance.

6.2.2 Perceived Pose Acquisition

The pose acquisition phase aim is to provide a suitable representation of the robot perceived position and orientation as required by the CLAM approach. With the adjective "suitable", we mean that the pose should be referred to the coordinate system of the previous robot pose. Indeed, when

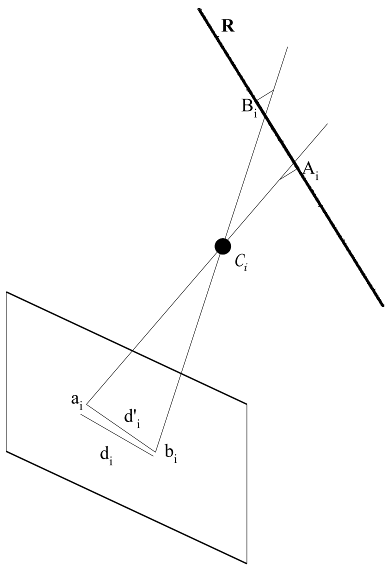


Figure 6.12: The interpretation lines and points A_i and B_i

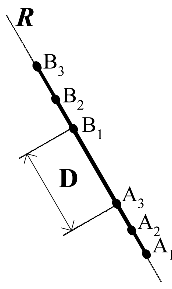


Figure 6.13: Determination of the extremes of 3D segment D

a robot moves, it really performs a displacement with respect to its previous position. For this reason, it is conceptually more correct thinking in terms of relative position with respect to the previous one and not to the absolute one. The odometric system should directly returns this displacement, i.e., the returned pose should be referred to the coordinate system of the previous robot pose.

When the odometric system does not return such kind of position, then some adjustments have to be performed. At this aim the information needed to perform this modification are the following:

- the current odometric robot pose, i.e., the robot position as returned by the odometric system;
- the previous odometric robot pose, i.e., the position in which the robot was before executing the displacement that lead it at the current position.

From the above considerations, to achieve a suitable perceived robot displacement, this phase needs the following steps to be executed (see figure 6.14):

1. *Odometry Acquisition*: it acquires the robot pose as provided by the odometric system. The result of this step is a robot pose called *odometric pose*;
2. *Perceived Robot Pose Computation*: if needed, it adjusts the odometric pose to provide a suitable representation of the perceived robot pose. At this aim, this step exploits the current odometric pose and the previous odometric pose to compute the rotational and translational parameters necessary to express the odometric pose in the the coordinate system of the previously perceived robot pose.

6.2.2.1 Odometry Acquisition

The robot is equipped with an odometric system continuously evaluating its pose. The pose is described by the pose vector in equation 6.2.

$$p_t = \begin{bmatrix} \Delta x & \Delta y & \Delta\theta \end{bmatrix}^T \tag{6.2}$$

where Δx is the displacement executed by the robot referred to the x axis, Δy is the displacement executed by the robot referred to the y axis, and $\Delta\theta$ is the rotation executed by the robot around the z axis.

The pose vector in equation 6.2 is referred to a *initial state* of the odometric system. An initial state represents, in every respect, a coordinate system. If the initial state is set once for all at the beginning of the robot exploration and it is not changed during robot operations, then each time the

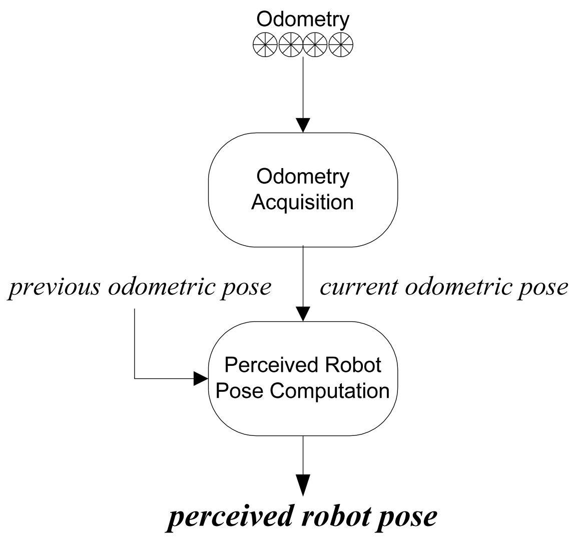


Figure 6.14: Perceived robot pose steps

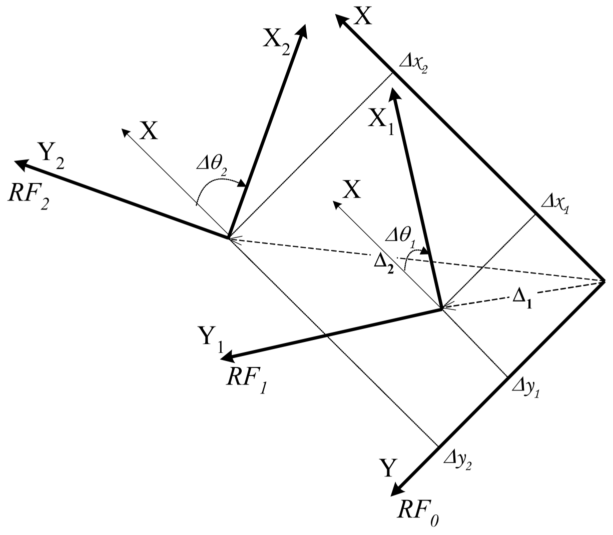


Figure 6.15: The absolute odometry

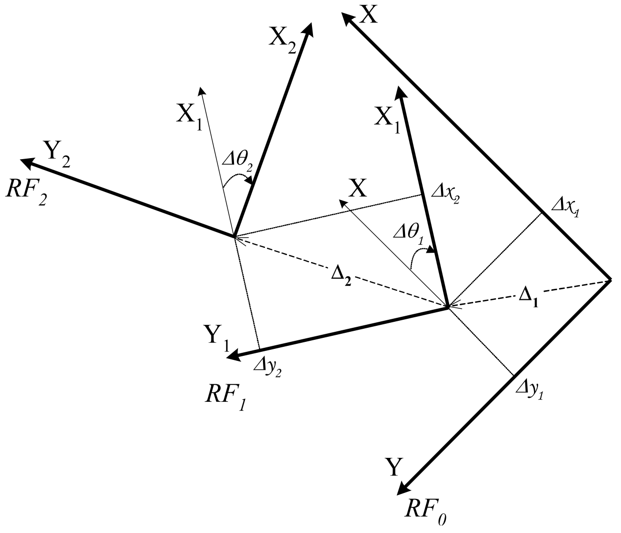


Figure 6.16: The delta odometry

odometric system returns the odometric robot position, it is always referred to this initial state. This situation is sketched in figure 6.15.

To achieve the desired robot perceived pose (i.e., referred to the previous perceived robot pose) as sketched in figure 6.16, this value must be reset at each movement. If it is not possible, then, knowing the initial state value, a transformation of the odometric pose is required.

This transformation is performed by the following step.

6.2.2.2 Perceived Robot Pose Computation

The aim of this step is to compute (if needed) the correct estimation of the translational and rotational parameters needed to derive the perceived robot pose (representing a displacement) from the odometric one (see figure 6.16). Then, this step, exploiting these computed parameters, calculates the perceived robot pose.

Since the robot can translate only on x and y axis, and can rotate only around the z axis, this step must compute the (x, y) translation parameters and the rotational parameter θ around z axis. This computation will also provide the rototranslation parameters that will be used in the Association phase of *Localisation* activity (see subsection 6.3.2).

The rototranslation parameters computation is made as follows.

Let be RF_1 the coodinate system realised by the previous odometric robot pose. RF_1 is known with respect to the odometry initial state (that realises the coordinate system RF_0). Concerning 6.3, let be O_1 the origin of the coordinate system RF_1 with respect to RF_0, and $\Delta\Theta_{10}$ the rotation angle around the z axis.

$$O_1 = \begin{bmatrix} \Delta X_{10} \\ \Delta Y_{10} \\ \Delta Z_{10} \end{bmatrix} \quad \text{and} \quad \Delta\Theta_{10} \tag{6.3}$$

Whereas, let be RF_2 the coordinate system realised by the current odometric robot pose. Likewise RF_1, RF_2 is known with respect to the odometry initial state too. Similarly (6.4), let be O_2 the origin of the coordinate system RF_2 with respect to RF_0, and $\Delta\Theta_{20}$ the rotation angle around the z axis.

$$O_2 = \begin{bmatrix} \Delta X_{20} \\ \Delta Y_{20} \\ \Delta Z_{20} \end{bmatrix} \quad \text{and} \quad \Delta\Theta_{20} \tag{6.4}$$

An antirototranslation of a point $p_2 = (x_2, y_2, z_2)$ from RF_2 to RF_0 is given by the matrix in 6.5.

$$\begin{bmatrix} \cos(\Delta\Theta_{20})x_2 - \sin(\Delta\Theta_{20})y_2 + \Delta X_{20} \\ \sin(\Delta\Theta_{20})x_2 + \cos(\Delta\Theta_{20})y_2 + \Delta Y_{20} \\ z_2 \end{bmatrix} \tag{6.5}$$

Whereas, a rototranslation of the same point p_2 from RF_0 to RF_1 is given by matrix in 6.6.

$$\begin{bmatrix} \cos(\Delta\Theta_{10} - \Delta\Theta_{20})x_2 + \sin(\Delta\Theta_{10} - \Delta\Theta_{20})y_2 + \\ cos(\Delta\Theta_{10})(-\Delta X_{10} + \Delta X_{20}) + \sin(\Delta\Theta_{10})(-\Delta Y_{10} + \Delta Y_{20}) \\ \\ -\sin(\Delta\Theta_{10} - \Delta\Theta_{20})x_2 + \cos(\Delta\Theta_{10} - \Delta\Theta_{20})y_2 + \\ sin(\Delta\Theta_{10})(\Delta X_{10} - \Delta X_{20}) + \cos(\Delta\Theta_{10})(-\Delta Y_{10} + \Delta Y_{20}) \\ \\ z_2 \end{bmatrix} \tag{6.6}$$

Simplifying 6.6 (neglecting z coordinates), we obtain the formula in 6.7.

$$\mathbf{R}(\Delta\Theta_{20} - \Delta\Theta_{10}) \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} + \mathbf{R}(\Delta\Theta_{01}) \begin{bmatrix} \Delta X_{20} - \Delta X_{10} \\ \Delta Y_{20} - \Delta Y_{10} \end{bmatrix} = \mathbf{R}_{21} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} + \mathbf{T}_{21} \tag{6.7}$$

Applying equation 6.7, we will be able to refer the current odometric robot pose with respect to the coordinate system realised by the previous odometric pose. The result is the perceived robot pose as required by CLAM representing the robot displacement.

Finally, likewise the segments, the robot perceived pose returned is enriched with a factor of uncertainty. The uncertainty is represented by a 3×3 symmetric matrix expressing the covariance.

6.3 Localisation Activity

Odometry is intrinsically error prone. Even if the errors are small at each movement, they are incrementally added so that we achieve a big error. In fact small errors (caused by effects such as drift or slippage), multiply overtime. Such effects are relatively easy to compensate if a model of the environment is readily available.

The aim of *Localisation* activity is to estimate the robot pose trying to reduce the errors produced by the odometric system. Exploiting the CLAM approach, key concept concerning *Localisation* activity is that it operates assuming a known environment. This means that the kind of information describing the environment that *Localisation* exploits for its purpose are supposed to be corrected (i.e., the perceived view is supposed to be correct).

To fulfill this goal, the robot pose estimation is computed exploiting the following kind of information as described in chapter 3:

- *current perceived located view*
- a *reference located view*

To achieve an estimation of the robot pose, as specified in chapter 3, *Localisation* activity is constituted by the following phases (see figure 3.2):

1. *Normalisation*: relates current perceived view and reference view to a common reference system;
2. *Association*: for each segment (geometrically located object) belonging to the current perceived view, looks for its homologous (i.e., representing the same real entity in the environment) in the reference view;
3. *Registration*: from the pairs of homologous segments, estimates the current robot pose.

6.3.1 Normalisation

Normalisation phase aim it to relate current perceived view and the reference view to a common reference system. This CLAM implementation relates current perceived view to the coordinate system of the reference view. Another solution would have been to make a sort of forecasting concerning the reference view: the reference view would be referred to the coordinate system of the current perceived view.

At the end of Normalisation, we will have two sets of 3D segments expressed in the same coordinate system.

Thus, the aim of this phase is to change the reference system of the current set of perceived 3D segments. The normalisation of each 3D segments is executed as described below.

Let be ${p^i}_1 = [x_1, y_1, z_1]^T$ and ${p^i}_2 = [x_2, y_2, z_2]^T$ the extremes of a segment $s^i = [x_1, y_1, z_1, x_2, y_2, z_2]^T$ belonging to the current perceived view. The current perceived robot pose is given by $p^i = [\Delta X, \Delta Y, \Delta\Theta]^T$. Both the perceived view and the perceived robot pose are the results of the *Perception* activity as described in section 6.2. Applying the rotation matrix 6.8 and the translation vector 6.9 we obtain in 6.10 the rototranslated segment ${s^i}_{rt}$.

$$\mathbf{R} = \begin{bmatrix} \cos(\Delta\Theta) & \sin(\Delta\Theta) & 0 & 0 & 0 & 0 \\ -\sin(\Delta\Theta) & \cos(\Delta\Theta) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos(\Delta\Theta) & \sin(\Delta\Theta) & 0 \\ 0 & 0 & 0 & -\sin(\Delta\Theta) & \cos(\Delta\Theta) & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.8}$$

$$\mathbf{T} = \begin{bmatrix} \Delta X \\ \Delta Y \\ 0 \\ \Delta X \\ \Delta Y \\ 0 \end{bmatrix} \tag{6.9}$$

$${s^i}_{rt} = \begin{bmatrix} \Delta X + \cos(\Delta\Theta)x_1 + \sin(\Delta\Theta)y_1 \\ \Delta Y - \sin(\Delta\Theta)x_1 + \cos(\Delta\Theta)y_1 \\ z_1 \\ \Delta X + \cos(\Delta\Theta)x_2 + \sin(\Delta\Theta)y_2 \\ \Delta Y - \sin(\Delta\Theta)x_2 + \cos(\Delta\Theta)y_2 \\ z_2 \end{bmatrix} \tag{6.10}$$

Concerning the segment uncertainty, it is updated when the segment is rototranslated. Let be:

- Λ_s^i the covariance matrix of segment s^i;
- Λ_p^i the covariance of the robot pose p^i;
- $DS_{XY\Theta}$ the matrix constituted by appending as columns the ΔX values, and ΔY values obtained from equations 6.9 (see equation 6.11);
- DS_{s^i} the matrix constituted by appending as columns x_1 , y_1, z_1, x_2, y_2, and z_2 values obtained from equations 6.9 (see equation 6.12).

$$DS_{XY\Theta} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ \begin{matrix}-\sin(\Delta\Theta)x_1+\\ \cos(\Delta\Theta)y_1\end{matrix} & \begin{matrix}-\cos(\Delta\Theta)x_1-\\ \sin(\Delta\Theta)y_1\end{matrix} & 0 & \begin{matrix}-\sin(\Delta\Theta)x_2+\\ \cos(\Delta\Theta)y_2\end{matrix} & \begin{matrix}-\cos(\Delta\Theta)x_2-\\ \sin(\Delta\Theta)y_2\end{matrix} & 0 \end{bmatrix} \quad (6.11)$$

$$DS_{s^i} = \begin{bmatrix} \cos(\Delta\Theta) & -\sin(\Delta\Theta) & 0 & 0 & 0 & 0 \\ \sin(\Delta\Theta) & \cos(\Delta\Theta) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos(\Delta\Theta) & -\sin(\Delta\Theta) & 0 \\ 0 & 0 & 0 & \sin(\Delta\Theta) & \cos(\Delta\Theta) & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (6.12)$$

Then, the new covariance for segment ${s^i}_{rt}$ will be given by equation 6.13.

$$\Lambda_{{s^i}_{rt}} = DS^T_{XY\Theta} \cdot \Lambda^i_s \cdot DS_{XY\Theta} + DS^T_{s^i} \cdot \Lambda^i_p \cdot DS_{s^i} \quad (6.13)$$

At the end of this phase, the current perceived view is geometrically expressed in the same reference system of the reference view. Consequently, we have two sets of 3D segments all referred to a common coordinate system.

6.3.2 Association

The aim of Association phase is to find pairs of homologous segments, i.e., pairs of segments belonging to different views, but representing the same segment in the real world.

The association may be realised exploiting different approaches all based on geometric constraints. Those approaches differ each other for the geometric features compared. Same examples may be found in [8], [8], [60].

The CLAM features are segments and we adopt an approach based on the minimum distance between segments.

The CLAM association mechanism is based on a simplified version of the Hausdorff distance [49]. Given two segments $\overline{A_1B_1}$ (belongin to the current perceived view) and $\overline{A_2B_2}$ (belonging to the reference frame), their minimum distance is given by:

$$min(\|A_2 - A_1\| + \|B_2 - B_1\|), (\|B_2 - A_1\| + \|A_2 - B_1\|) \quad (6.14)$$

Since we need univocal pairs of segments, for those that do not satisfy this constraint, it is chosen the pairs that have the minimum distance. For this version of CLAM, the segments uncertainty is not taken into account.

The problem of mismatch (i.e., the problem of relating wrong segments) is solved setting a threshold for the distance.

As result of this phase, we have a set constituted by pairs of 3D segments representing the real segment.

6.3.3 Registration

This phase constituites the core of the *Localisation* activity, since it provides an estimation of the correct robot pose exploiting the results achieved by the phases previously explained. Referring to chapter 3, CLAM approach is based upon the consideration that *Localisation* activity estimates the robot pose inside an environment that is assumed known.

For this reason, if the sets of segments that have been associated in the previous phase do not fit perfectly (they are assumed correct), then this error should be caused by the erroneous robot perceived pose.

Then, the aim of this phase is to correct the pose provided by the odometry system by finding the best rototranslation parameters between current perceived and reference views that realises the best fitting/association of the segments.

The technique used in CLAM is an iterative one based upon the *Monte Carlo* method. The method iteratively evaluates the new distance of the segments pairs each time new rototranslation parameters describing the potential robot pose are generated in a randomatic way. Finally, the rototranslation that have produced the minimum distance is selected as the best estimation of the robot pose.

At the end of the *Localisation* activity, we have deduced an estimation of the robot pose that corrects the one returned by the odometric system. Exploting CLAM principles, the computation of the estimated pose has been made upon the condition that the information provided by the trinocular stereo system is correct.

Last consideration concerns the iterativeness of some of the phases constituting the *Localisation* activity: the "Normalisation", "Association", and "Registration" phases may be iterated to affine the estimation of the robot pose. In this case, the input for the "Nornalisation" activity is the result of the "Register" activity.

6.4 Modelling Activity

The *Modelling* activity consists in incrementally updating the map of the environment the robot is exploring.

As the *Localisation* activity assumes the correctness of the reference located view and tries to correct the pose, the *Modelling* activity considers the pose correct and tries to adjust the visual information.

This activity, when executed, takes into account all the views acquired form the last time the activity has been performed.

The views that will update the *world model* are all referred to the absolute reference frame. Our approach to map building is extremely selective since it introduces in the *world model* only those segments that have been identified in at least two views. At the first run, this seems to be reductive, but we should not forget that the robot is exploring an unknown environment. For this reason, it moves slowly and covers a brief distance at each movement. It is highly probable that what it sees at time t, it will see again at time $t+1$. Adding in the *world model* segments that have been never matched, will lead to introduce something that really does not exist: with high probability those unmatched segments derive from, for instance, an error in the *Perception* activity.

For the above reasons, the *Modelling* activity incrementally constructs the *world model* merging *only* those segments that have been associated in the Association phase of *Localisation* activity.

At this aim *Localisation*, for each associated segments, keeps an history of previously associations of the same segment. To explain this concept, an example will be provided. Suppose that at time t the segment s^t belonging to current perceived view has been associated to segment s^{t-1} perceived at time $t-1$. The *Localisation* activity (at time $t+1$) associates the segment s^{t+1} with the segment s^t. This means that s^{t-1}, s^t, and s^{t+1} are all the representation of the same real segment.

This history of associations is related to the reference frame of the *world model*. The decision of submitting the rototranslation of segments in the absolute reference frame to the *Localisation* activity is motivated by the consideration that positioning issues concern only *Localisation*. We apply what is called separation of concerns: the management and the knowledge of robot pose is matter of *Localisation*, whereas the management and the construction of the *world model* is matter of *Modelling*.

From the consideration above described and referring to figure 3.3, *Modelling* activity consists in the *Fusion* and *Integration* phases.

6.4.1 Fusion

Localisation activity, after the updating of the robot pose, refers the associated segments to the absolute reference frame (i.e., the reference frame of the *world model*) and keeps track of a linked list of associated views (i.e., subsets of the views containing also the associated segments).

The Fusion phase fuses together all the homologous segments belonging to the linked list and adds the resulting segments in the *world model*.

The fusion operation estimates the position of the real segment and updates its uncertainty. In doing so it exploits equation 6.15 to determinate the new covariance matrix, and equation 6.16 to estimated its current position. This is a simplification form of Kalman filter.

$$\Lambda_{\hat{s}_M} = (\Lambda_{s_t}^{-1} + \Lambda^{-1}{}_{s_{t-1}})^{-1} \tag{6.15}$$

where Λ_{s_t} is the covariance matrix of the segment in the perceived view acquired at time t, and $\Lambda_{s_{t-1}}$ is the covariance matrix of the segment in the perceived view acquired at time $t-1$.

$$\hat{s}_M = \Lambda_{\hat{s}_M}(\Lambda^{-1}{}_{s_t} s_t + \Lambda^{-1}{}_{s_{t-1}} s_{t-1}) \tag{6.16}$$

where s_t is the segment in the perceived view acquired at time t, and s_{t-1} is the segment in the perceived view acquired at time $t-1$.

6.4.2 Integration

This phase deals with the integration of the fused segments in the *world model*. Integration means that the segments are someway introduced in the *world model*:

- if the fused segment is a new segment, then it is simply added to the *world model*;
- if the fused segment corresponds to a pre-existent segment in the *world model*, than the two segments are fused exploiting the technique used by the Fusion phase.

To verify if a fused segment reifies a real segment whose representation is already presents in the world model, an association mechanism is exploited. We use the same exploited in the association phase of *Localisation* activity.

Chapter 7

RTP Exploitation for CLAM

7.1 Introduction

CLAM approach is based upon the *concurrent* execution of *Localisation* and *Modelling* constituting the two main activities a robot executes when exploring an unknown environment with uncertain information about its pose.

As described in chapter 3, even if *Localisation* and *Modelling* are related, they act on different time scales so that they may be considered mostly independent one another. When some kind of synchronisation in needed, a criticality arises. A suitable Strategy, relying on the observation of the criticalities, adjusts the relative rates of activities to meet as soon as possible the synchronisation requirements.

Statements like "different time scale", "...some kind of synchronisation ...", "...suitable strategy ...", "...observation of criticalities ...", "...adjusts the relative rates ...", recall concepts adressed in Real-Time Performers. Time scales, Strategy, dynamic temporal management at the application level are the core concepts of the Real-Time Performers architecture (see chapters 4 and 5). Real-Time Performers allows to build systems in which different timings drive the execution of the activities making the system itself. RTP operates a separation of concerns between information processing (both elaboration and information) and policies driving (strategy) the execution timing of the processing. Finally, RTP reifies a set of suitable architectural abstractions modelling the temporal behaviour of a computational system.

However, the major contribution given by RTP exploitation in CLAM is the management of criticalities. Since a criticality may be solved by adjusting the activities relative execution speeds, and RTP provides the appropriate mechanisms, then the RTP is fits perfectly for the design and implementation of CLAM systems.

Figure 7.1 sketches how the concepts related to the the CLAM approach

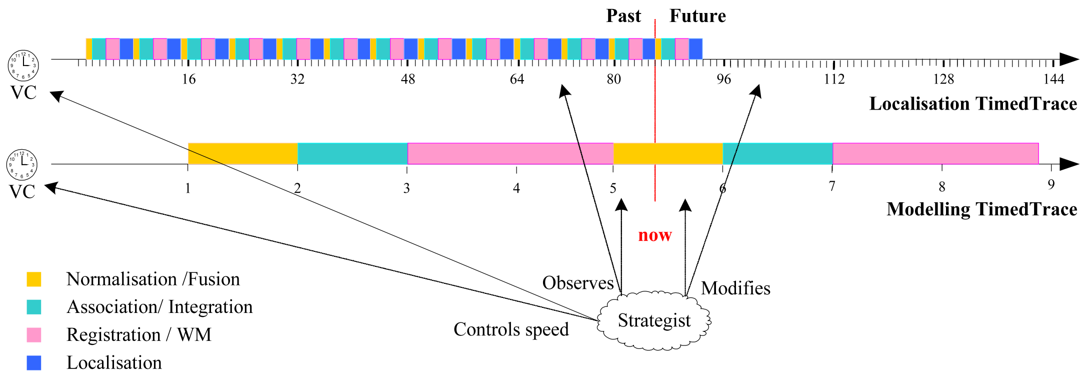


Figure 7.1: The RTP exploitation for CLAM

are reified through the RTP ones.

The chapter is organised in two main sections: the first describing the system structure (information types, i.e., visibles, and computational components, i.e., Performers), whereas the second describing system dynamics. In this section we will present both system normal behaviour (concurrence), and system criticalities management (synchronisation).

In the following, several UML class diagrams will be presented. The convention used is that in general if a class belongs to a different package, it will presented with no detailed information. Moreover, if the class belongs to the RTP framework, it will be filled with blue colour, whereas if the class belongs to a different package defined in CLAM, it will be filled with yellow colour.

7.2 General System Structure

Since a self-localisation and mapping reconstruction system for autonomous robot exploration is quite complex, to not bore the reader, only main concepts will be provided in the sequel.

The design of the system has been made trying, when possible, to opportunely separate system information, computational entities, and execution policies defining CLAM.

Consequently, a prior main separation (see figure 7.2) is made between system data (`type` package), computational components (`performers` package), and strategy (`strategy` package).

`types` package contains classes that (directly or indirectly) constitutes the Visible variables as defined in chapter 4; `performers` package contains classes that define CLAM Performers; `strategy` package contains classes

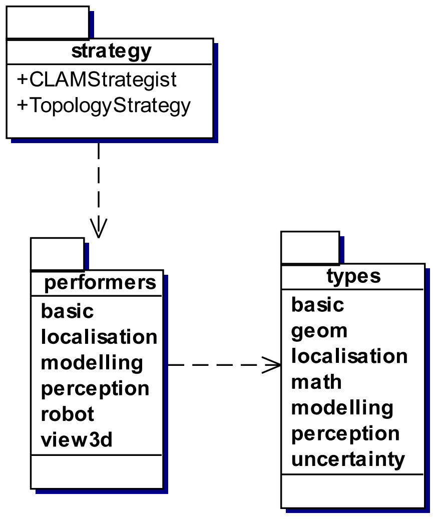


Figure 7.2: The main packages

for describing system dynamics.

Section 7.3 deals with the classes defining the information in the CLAM system, section 7.4 deals with the Performers defined for CLAM purpose, and section 7.5 deals with the system dynamic behaviour.

Referring to chapter 3, three major activities define CLAM system: *Perception*, *Localisation*, and *Modelling*:

- *Perception* deals with information acquisition (and pre-elaboration);
- *Localisation* deals with robot pose estimation;
- *Modelling* deals with environmental map reconstruction.

`types` and `performers` packages are organised into sub-packages reflecting the separation between the CLAM activities as above remarked.

7.3 Information

The classes of information exploited by the CLAM system are arranged in the packages sketched in figure 7.3.

Packages relevant for our discussion are the `basic`, teh `perception`, the `localisation`, and the `modelling` ones. The others constitute the basis for their definition. For clarity purpose, a very brief explanation of the classes these basic packages comprise will be given:

- `geom` package addresses geometrical issues providing the following classes:

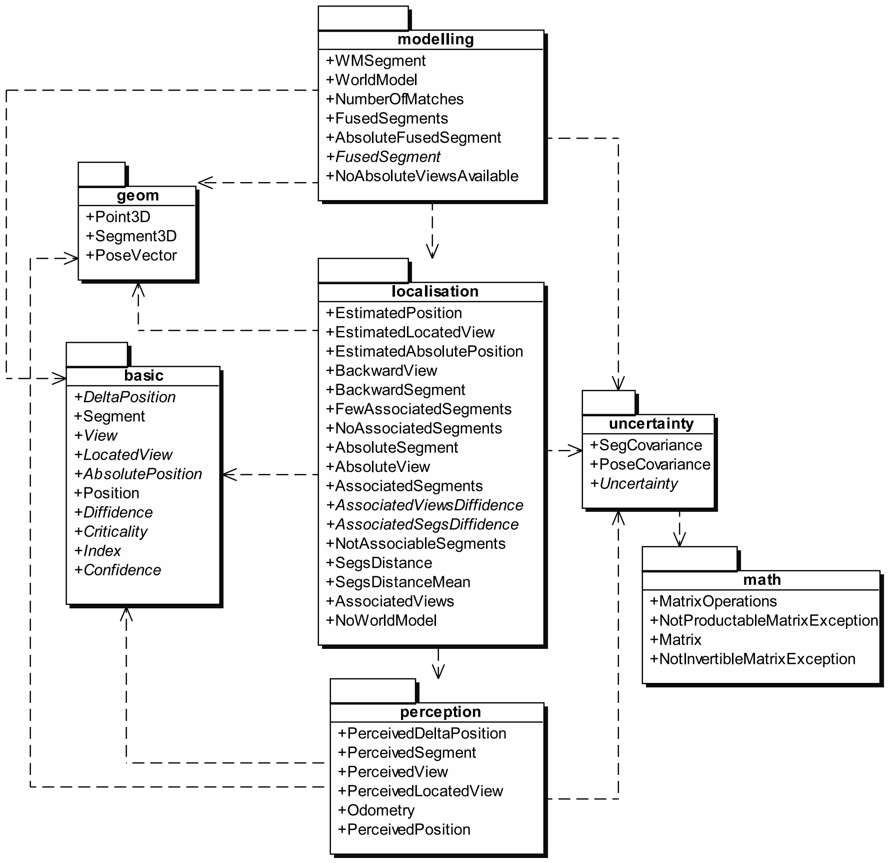


Figure 7.3: The packages constituting types

  - `Point3D` modelling a generic point in a 3D space;
  - `PoseVector` class modelling a translation over the (x, y) plane and a rotation around the z axis. An instance from this class models a generic robot position;
  - `Segment3D` class modelling a segment in a 3D space. As explained in section 6.2.1, one of the results of *Perception* activity (concerning trinocular stereo system), is a 3D view made up by 3D segments. A 3D segment is defined by a pair of 3D points (its extremes). Beside the others, the class publishes two interesting methods: `rototranslate(PoseVector)`, and `antirototraslate(PoseVector)`. The first method rototranslates the 3D segment with respect to the rototranslation parameters in input. The result is a 3D segment referred to another coordinate system corresponding to a previous (temporally speaking) robot position. The second one rototranslates a segment too, but this rototranslation may be view as a forecast of its position: given a 3D segment referred to a coordinate system generated by the robot position at time t, the method computes the coordinates of the 3D segment supposing that the robot at time $t + 1$ will be in the position given by the input argument;

- `uncertainty` package contains classes modelling the imprecision of the perceptive devices, of the image processing algorithms, and of the errors in estimating both robot pose and segments belonging to the world model. The uncertainty is expressed by means of matrixes. `SegCovariance` models the matrix covariance of a 3D segment, whereas `PoseCovariance` models the matrix covariance of the robot pose;

- `math` package provides classes exploited in the `uncertainty` package. Since uncertainty is represented in a matrix form, this packages defines `Matrix` class and a set of classes supporting the operation between matrixes.

7.3.1 Basic types

The `basic` package deserves more discussion since it contains basic classes from which the information exploited by the three activities is built.

The overall organisation of the package is sketched in figure 7.4.

As emphasises by the italic typeface used for the name of the classes, the concepts here described are abstract, i.e., they do not represent concrete entities. They model general concepts that will be specialised by concrete representations. As mentioned in chapter 1, the aim of this CLAM implementation is to verify the validity of the approach. Maintaining basic

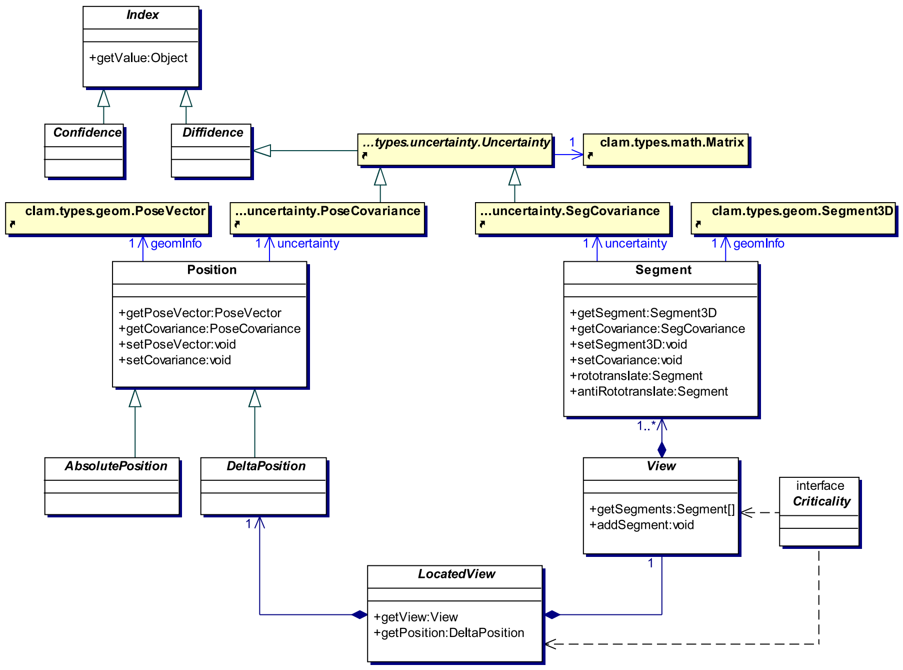


Figure 7.4: The classes of the `basic` package

methods and classes as general as possible, it guarantees their reuse for different implementations of *Localisation* and *Modelling* exploiting CLAM approach.

- `Index` class models a general estimation value of reliability. An `Index` is associated to every kind of information the value of which is subjected to satisfy some kind of constraint. This class is general enough to represent both the goodness and the badness of the information to which it is associated. The only abstract method it publishes is `getValue()` returning the actual value of the index.

- `Confidence` class defines the level of plausibility of the entity to which the confidence is referring to. The higher is the confidence value, the higher is the level of reliability of the entity to which it belongs. On the contrary, the lower is the confidence value, the lower is the level of reliability of the entity to which it belongs.

- `Diffidence` class defines the level of plausibility of the entity to which the diffidence is referring to. The lower is the diffidence value, the higher is the level of reliability of the entity to which it belongs. On the contrary, the higher is the diffidence value, the lower is the level of reliability of the entity to which it belongs. Examples of Diffidence values are the `PoseCovariance` and the `SegCovariance` classes: they defines the level of confidence of a position and a segment respectively. `Diffidence` is a general concept that must be opportunely reified;

- `Position` class defines a general position in a 3D space. A position is completely defined by rotational and translational parameters (reified by the association with `PoseVector` class), and by its level of uncertainty (reified by the association with the `PoseCovariance` class). A position specifies where the robot is, i.e., it specifies the robot pose with respect to a general coordinate system. This definition is general enough, since no assumption is made about the coordinate system is used. `Position` class reifies the *robot pose* P defined in section 3.2 by equation 3.1;

- `AbsolutePosition` and `DeltaPosition` classes are special kind of positions (they are subclasses of `Position` class) representing respectively the robot pose with respect to an absolute coordinate system and to a relative one. `DeltaPosition` conceptually represents the displacement the robot make at each movement;

- `Segment` class models a general 3D segment as handled by CLAM activities. What we mean is that a pure geometric information about a segment is not sufficient to accurately perform *Localisation* and *Modelling* activities: a `Segment` instance is not only defined by geometrical information (reified by the association with `Segment3D` class), but is enriched by uncertainty (reified by the association with `SegCovariance` class) specifying the confidence level of the segment geometry and existence. No assumption is made about the typology of segment: it may be a segment perceived by the robot, a segment of the world model, the segment of the reference view, and so on. `Segment` class reifies a *geometrically located object* glo_i defined in section 3.2;

- `View` class is defined as an aggregation of `Segment` instances. It models a general view. No assumption is made about the origin of the view, i.e., it may represents exactly the view perceived by the robot, or a view referred to another coordinate system, or the world model, and so on. The concept here described is simply that a set of someway related segments are grouped to form a view. `View` class reifies the *view* V defined in section 3.2 by equation 3.2;

- `LocatedView` class associates a view to the position to which is referred. As sketched in figure 7.4, `LocatedView` class is associated with `DeltaPosition` class instead of `Position` class. This is justified by the fact that for related view we intend views whose coordinate system is not the absolute one. `LocatedView` class reifies the *located view* LV defined in section 3.2 by equation 3.3;

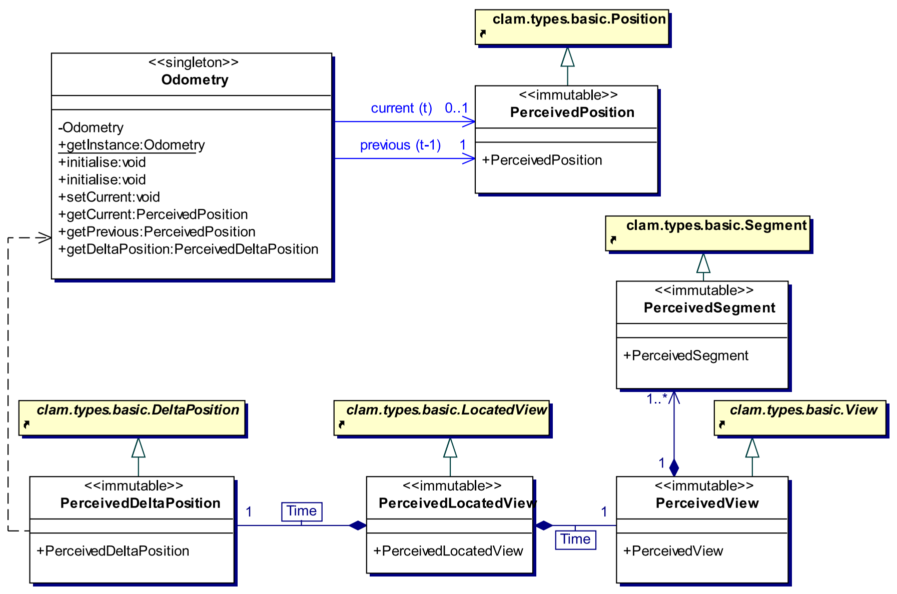


Figure 7.5: The classes of the `perception` package

- `Criticality` interface is a key concept in CLAM. This class models any kind of information useful to drive opportunely CLAM execution. Referring to chapter 3, this class represents the information that is constantly under control when performing activities in CLAM. Precisely, this is the information exploited to plan correctly the beaviour of the system. If a criticality arises, then an opportune strategy should be adopted. This means that if a criticality arises, then a notification is made to communicate that a greave situation is occurred. This class may be think as modelling an exception that may occur during execution. This is a general concept that must be opportunely reified.

7.3.2 Perception types

Perception activity exploits classes of information belonging to this package (see figure 7.5). All the classes belonging to this package have assigned the $<< immutable >>$ stereotype. This property means that, when instances are created, they do not change own state during their lifecycle. This is an important property that help in maintaining well distinct the information types exploited during the CLAM activities execution. The perception types are the following:

- `PerceivedPosition` class describes a robot position as perceived by the odometric system. Since the odometric system may return both an absolute position or a relative one (see subsection 6.2.2.1), this class is a general position (i.e., it is a specialisation of the `Position` class);

- `Odometry` class models the robot odometry. It represents the odometry state. The state is expressed by a current acquired position and a previous one (in temporal sense). These concepts are modelled by the two associations that this class has with the `PerceivedPosition` class.

- `PerceivedDeltaPostion` class models the displacement the robot executes at each movement. The class specialises the `DeltaPosition` one, and, consequently, it reifies the rototranslation parameters that must be used to refer actual robot pose to another one. `PerceivedDeltaPostion` class reifies the *perceived pose* PP defined in section 3.2 by equation 3.4;

- `PerceivedSegment` class represents the 3D segment as perceived by the three cameras. A `PerceivedSegment` is a particular type of `Segment`. This class reifies a *perceived geometrically located object* $pglo_i$ defined in section 3.2;

- `PerceivedView` class models the view perceived by the robot. This is the result of the *Perceived View Acquisition* phase described in section 6.2.1. This kind of view (it is a specialisation of the more general `View` class) is composed by the set of perceived segments (i.e., instances of `PerceivedSegment` class). `PerceivedView` class reifies the *perceived view* PV defined in section 3.2 by equation 3.5;

- `PerceivedLocatedView` class models a complete *Perception*, i.e., it comprises all the perceptive inputs. For this reason a `PerceivedLocatedView` is a view referred to the robot pose (as a matter of fact it is a subclass of the `LocatedView` class). It is fundamental the association class named `Time` presents in the associations `PerceivedLocatedView`, `PerceivedDeltaPosition` and `PerceivedLocatedView`, `PerceivedView`. This is a constraint indicating that a *Perception* is made up by perceptive inputs taken at the very same time. `PerceivedLocatedView` class reifies the *perceived located view* PLV defined in section 3.2 by equation 3.6.

7.3.3 Localisation types

Localisation activity exploits the classes of information belonging to this package.

To provide a clear description of the classes in this package, we will subdivide them with respect to the *Localisation* step they support.

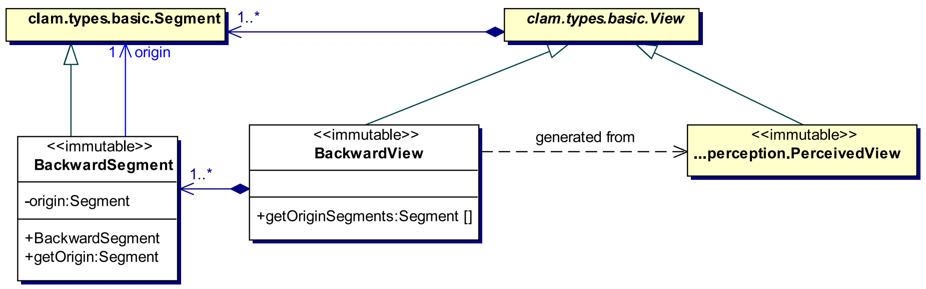


Figure 7.6: Types concerning *Normalisation*

What is said for << *immutable* >> stereotype in *Perception*, still holds here. The use of the << *mutable* >> stereotype means that the instance state may change during system execution.

7.3.3.1 Types concerning *Normalisation*

Figure 7.6 sketches the types produced by the Normalisation phase. Referring to section 6.3.1, Normalisation changes the coordinate system of a set of segments (a view). In particular, this phase refers the current perceived view to the coordinate system of the reference view. Since the reference view is a view that is referred to a previous robot pose (just the previous one, or the starting one), we call the resulting view `BackwardView`. This view is made up by `BackwardSegment`s: special kind of segments preserving their original position.

7.3.3.2 Types concerning *Association*

Association plays a crucial role in CLAM since the result of its execution may cause a change in system behaviour (i.e., from concurrent to synchronised). Consequently, some of the types defined for its purpose (sketched in figure 7.7), are, as we will see later in section 7.5, fundamental for the CLAM activities right timing detection.

The aim of Association phase (see section 6.3.2) is to generate pairs of homologous segments (i.e., representing the same real entity) from two different views. `AssociatedSegment` class reifies those pairs. It contains a pair of segments: one belongs to the actual perceived view, the other belongs to the reference view. `AssociatedViews` class contains all the homologous segments.

As sketched in figure 7.7, the goodness of a pair of associated segment is given by the `AssociatedSegsDiffidence` class, whereas the goodness of all pairs of associated segments is given by the `AssociatedViewsDiffidence`

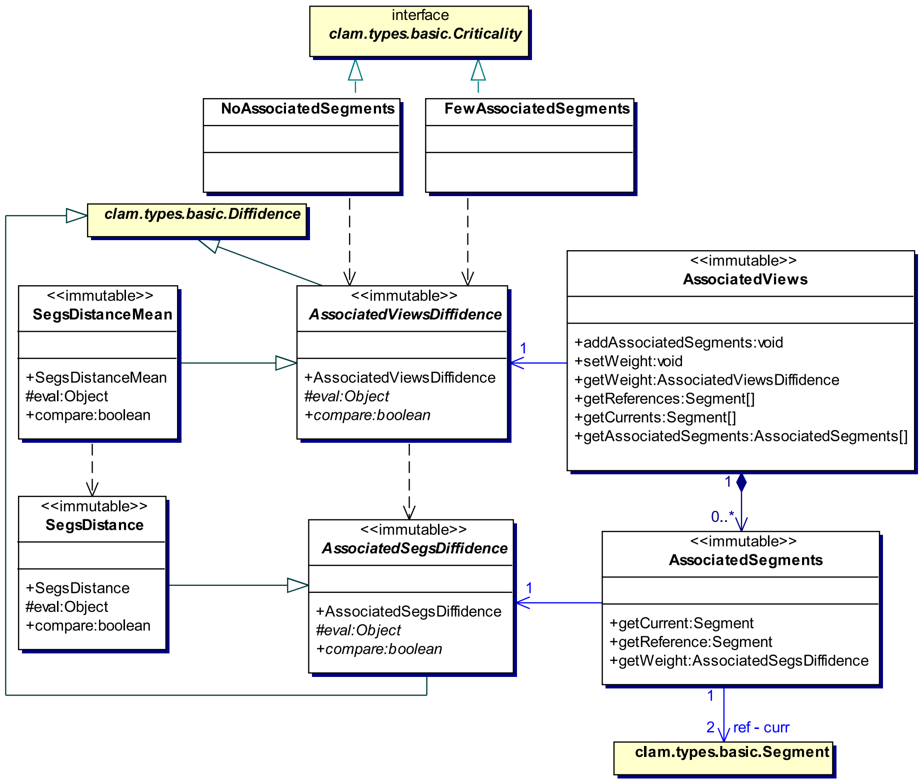


Figure 7.7: Types concerning *Association*

class. These classes are special kind of `Diffidence`: they describe the quality of each pair of segments and the quality of the set of pairs respectively. The lower are their values, the better are the associations.

Since `AssociatedViews` depends on the set of `AssociatedSegments`, the same is true for its diffidence value, i.e., `AssociatedViewsDiffidence` is evaluated from the values of each `AssociatedSegsDiffidence`. Finally, these classes are defined abstract for generality purposes.

The system we have realised, exploits the following concrete reification: `SegsDistance` and `SegsDistanceMean`. The former represents the geometrical distance between the pair of segments, the latter represents the mean value between all the distances.

Referring to subsection 3.4.1, a criticality during the execution of *Localisation* activity may occur when the reference view is not plausible. What we mean is that the reference view does not represent the same environment the robot is perceiving. This situation causes the failure of the association phase implying an high value of the `AssociatedViewsDiffidence`. Criticalities depend on the value of this confidence value. We have identified two kind of criticalities: `NoAssociatedSegments` and `FewAssociatedSegments` representing respectively the situation in which no segments have been associated (worse case), and few segments belong to both the views. When one of the two situation takes place, the corresponding criticality arises and strategy must adopt a suitable corresponding policy.

7.3.3.3 Types concerning *Registration*

The information produced by Association (`AssociatedViews`) is then elaborated by Registration phase with the aim of estimating robot pose.

The evaluated estimated pose is described by `EstimatedPosition` class. This is a special kind of `DeltaPosition`. The reason for this choice is that, despite the nature of the reference view, a robot, when moving, performs a displacement. If the reference view is the previous view, then actual `EstimatedPosition` is the displacement the robot performs from its previous position to its current one. Even if the reference view is the world model, the robot actual `EstimatedPosition` may be view as the result of a displacement that leads the robot to be in its actual position from its initial one. `EstimatedPosition` class reifies the estimated pose EP defined in section 3.2 by equation 3.10.

7.3.3.4 Types produces at the end of Localisation

The information generated at each *Localisation* step are sketched in figure 7.8.

- `EstimatedLocatedView` class relates a perceived view (`PerceivedView`) to an estimated robot position (EstimatedPosition). This class (as we

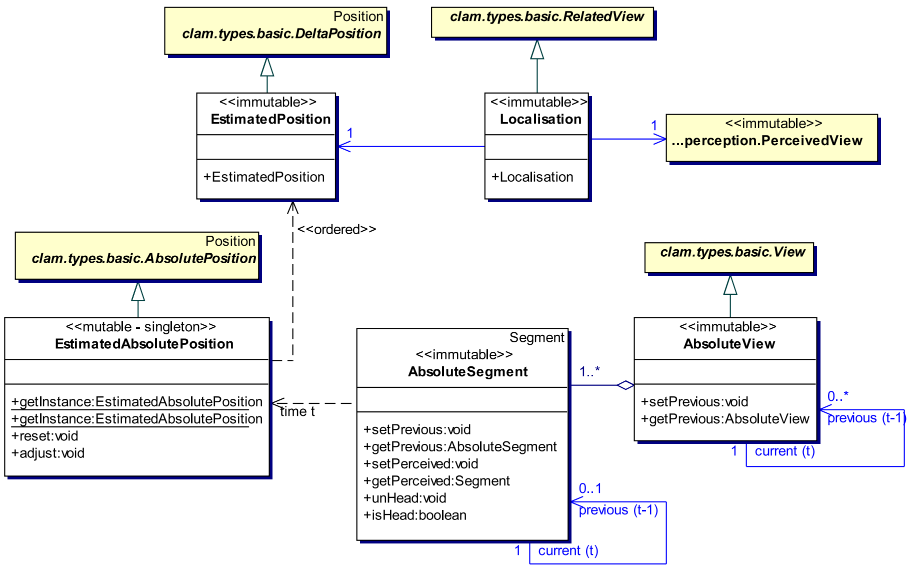


Figure 7.8: Basic types concerning *Localisation*

will see later) is fundamental for next *Localisation* activity. The class reifies the estimated located view *ELV* defined in section 3.2;

- `EstimatedAbsoluteView` class reifies the absolute pose of the robot. This class is a $<<$ *singleton* $>>$. It is also a $<<$ *mutable* $>>$, since its value changes each time a new `EstimatedPosition` is available.
- `AbsoluteSegment` class models a segment (it is a subclass of the `Segment` class) referred to the origin of the world model. Note that an `AbsoluteSegment` is in association with another `AbsoluteSegment`. This association is needed to keep track of the associated segments;
- `AbsoluteView` class is a set of `AbsoluteSegments`. It is a special kind of `View`. As for the segments, even `AbsoluteViews` are each other linked: the current created `AbsoluteView` is linked to the previous `AbsoluteView`.

7.3.4 Modelling types

Classes describing *Modelling* activity types are very simple as shown in figure 7.9.

Main concepts are `WMSegment` class, modelling a segment in the world model (referring to section 3.2, it is a *world model geometrically located object*), and `WorldModel` class, defining the world model (referring to section 3.2, it is defined by equation 3.11). The only interesting method in

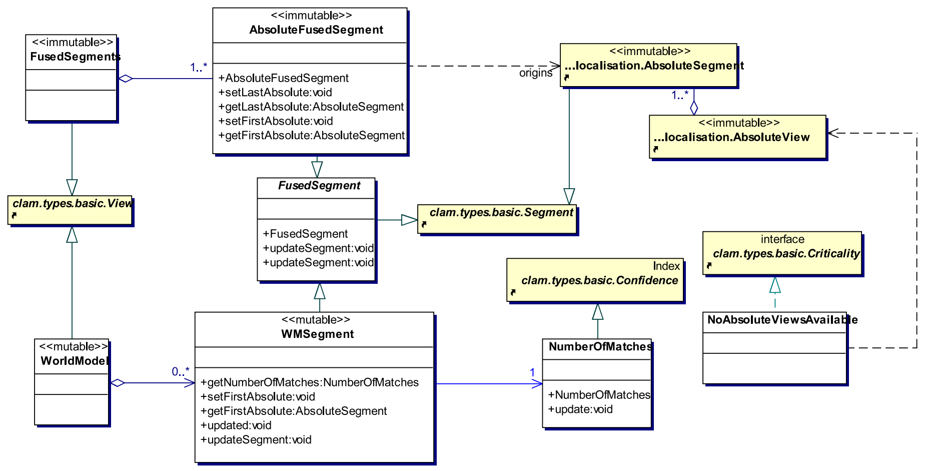


Figure 7.9: The classes of the `modelling` package

`WMSegment` class is `updateSegment()` that is used to modify (the class is $<<$ *mutable* $>>$) the segment exploiting new information about it.

A `WMSegment` is characterised by a index of quality called `NumberOfMatches`: it specifies how many times this segment has been perceived during exploration. This is an important parameter to test the validity of a segment, e.g., if the `NumberOfMatch` level of a segment is low, then the probability that this segment really does not exist is high.

Segments belonging to world model are generated (and modified) from `AbsoluteFusedSegment` instances. These instances are the result of the fusion phase. In turn, `AbsoluteFusedSegments` are derived from `AbsoluteSegments` created by *Localisation* activity. Absolute segments are arranged inside their view called `AbsoluteView`. If there is not a plausible history of this kind of views, then a criticality arises. An example of plausibility criteria concerns the availability of an updated set of views (`NoAbsoluteViewsAvailable`).

7.4 Performers

Performers are structured into packages likewise types. All the packages are sketched in figure 7.10.

This subdivision into packages has the aim of maintaining logically grouped Performers dealing with the same problematic:

- `basic` package contains fundamental classes for criticality evaluation and notification;
- `view3d` package contains all the classes needed to graphically visualise the system state. The visualiser Performer is `Viewer3D` class, the

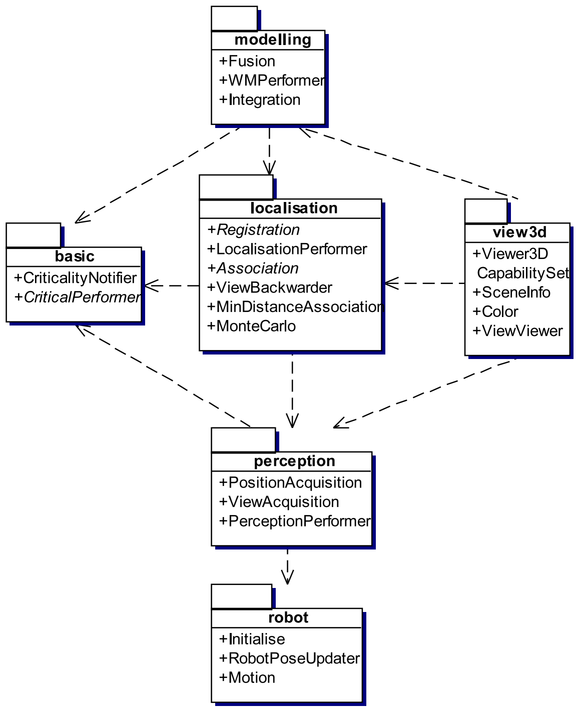


Figure 7.10: The performers packages for CLAM

other classes are used by the Performer itself. Since the role of this package is not fundamental for our thesis, it will not be described in the following.

- `robot` package contains Performers dealing with the robot. `Initialise` Performer aim is to change setting parameters for the robot. The `Motion` Performer deals with the motion of the robot. Whereas the `RobotPoseUpdater` deals with the update of the odometric position value (if necessary). This package will not be described in the following for the same reason involving `view3d` package;

- `perception`, `localisation`, and `modelling` packages group all the Performers needed for *Perception*, *Localisation*, and *Modelling* purposes respectively. These packages will be described in detail since they constitute the core of the *Localisation* and *Modelling* activities.

7.4.1 Basic Performer

The basic package contains classes for criticality notification and evaluation.

The `CriticalPerformer` class defines a special kind of Performer. It is a classical command acceptor (as described in sections 4.3 and 5.3.3), but it also evaluates if some kind of criticalities arises from the execution of one of its acceptable commands (method `evalCriticality()`).

When a criticality occurs, the `CriticalPerformer` writes it (an instance of the `Criticality` class) on the corresponding `CriticalityNotifier` out visible. `CriticalityNotifier` is a special kind of `NotifyingVisible` the state of which is given by its `Criticality` instance.

`CriticalityNotifier` out visibles are notifying entities since their values must be immediately known to perform the opportune strategy.

7.4.2 Perception activity

Performers dealing with *Perception* activity are sketched in figure 7.12.

`PositionAcquisition` Performer is in charge of acquiring current robot pose. This Performer reifies the *Perceived Pose Acquisition* phase as described in subsection 6.2.2.

At this aim, the only acceptable command by the Performer is the `ACQUIRE_POSITION` one. When it receives this command, it asks the robot to return its perceived pose.

The result is a `PerceivedDeltaPosition` specifying the last displacement executed by the robot, as described in subsection 7.3.2.

`ViewAcquisition` Performer is in charge of acquiring current perceived views. The only acceptable command for this Performer is the `ACQUIRE_VIEW` one. This command, when executed, asks the robot to return a triplet of

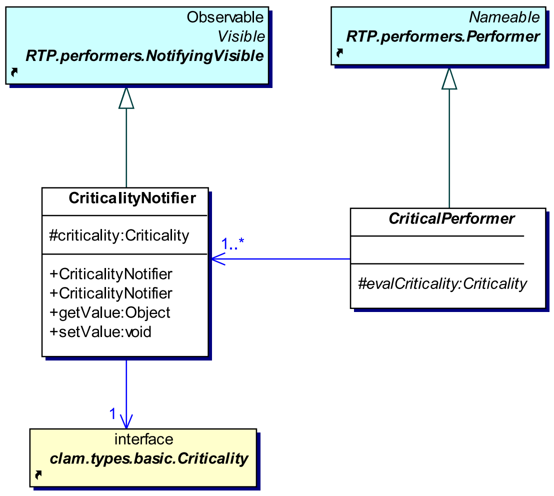


Figure 7.11: The basic package

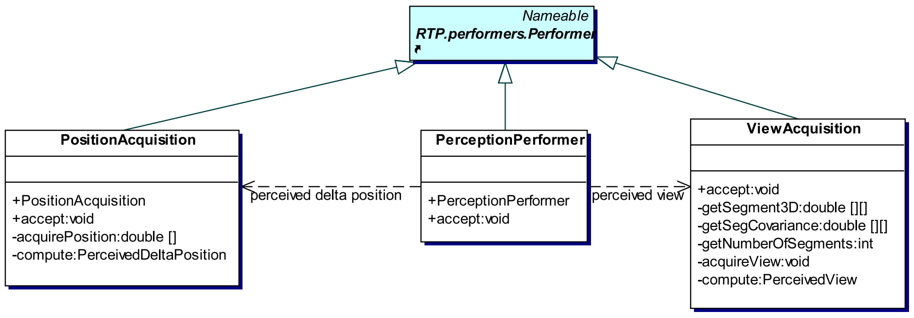


Figure 7.12: The Performers for *Perception* purpose

images, one from each camera, and reconstructs the current perceived 3D view as described in subsection 6.2.1.

After the execution of the command, the current perceived view is available and completely described by the `PerceivedView` class (see subsection 7.3.2).

Both `PositionAcquisition` and `ViewAcquisition` Performers are wrapper classes since the concrete execution of both pose and view acquisition are in charge of a pre-existent application (named *Trinocular*) developed using the C++ programming language.

Finally, the aim of the `PerceptionPerformer` is to receive and to merge the information (*visibles*) provided by the other Performers. The execution of the acceptable command `PERCEPTION`, let the `PerceptionPerformer` to generate a `PerceivedLocatedView` instance (as described in subsection 7.3.2) containing all the perceptive inputs.

`PerceivedLocatedView` instances constitute the information provided by the *Perception* activity to the other activities.

7.4.3 Localisation activity

Each phase of the *Localisation* activity is carried on by a specific Performer (see figure 7.13).

For generality purposes, some of the Performers are left abstract to be used under different implementation. As specified, the algorithms described in chapter 6 constitute one of the possible implementations.

`ViewBackwarder` Performer reifies the Normalisation phase as described in subsection 6.3.1. The Performer recognises only the `BACKWARD` command. To be executed, the command needs that the following visibles are available: the view to backward (`ViewToBackward` in visible), and a robot pose in which to refer the view (`BackwardPosition` in visible).

A new view is generated at the end of the performed command. This view (`BackwardView`) is made available by means of the out visible named `BackwardView`.

`Association` Performer aim is to find pairs of homologous segments belonging to two different views (the current perceived and the reference one). The operation may be performed if the views (i.e., the sets of segments) are referred to the same coordinate system. This is the reason why this Performers uses a `BackwardedView` as input for computation. This view is received from `BackwardPerformer` by means of the in visible `BackwardView`. The reference view is maintained by `ReferenceView` in visible. The result of the execution of `ASSOCIATE` command (the only acceptable one) is a sets of associated segments (a `AssociatedViews` instance, see section 7.3.3).

`Association` Performer is a `CriticalPerformer`: at the and of the association mechanism, it verifies the occurrence of some kind of criticality. Criticalities may occur if the reference view is not a plausible representation

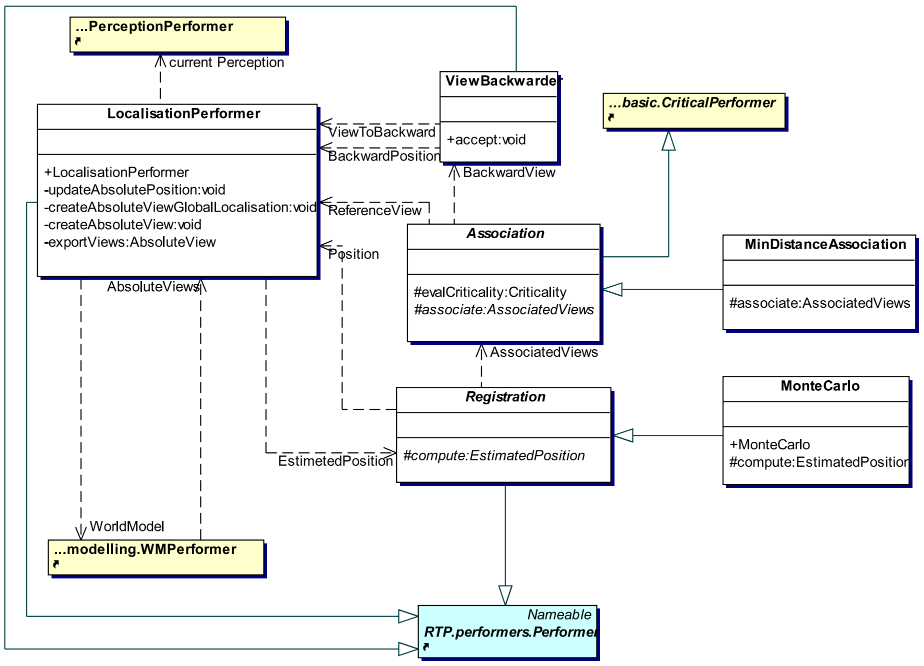


Figure 7.13: The Performers for *Localisation* purpose

of the environment the robot is perceiving. When this situation occurs, the Performer updates its notifying out visible (`CriticalityNotifier`) named `Association` with a value representing the kind of criticality. Contrary, if the Association is performed in a successful way, then the evaluated `AssociatedViews` is stored in the `AssociatedViews` out visible.

This implementation of the association algorithm is intentionally defined abstract to allow the exploitation of this Performer under different association mechanisms.

We have defined a `MeanDistanceAssociator` class to perform association. This class computes the geometrical distance between segments as described in subsection 6.3.2.

The aim of the `Registration` Performer is to estimate robot pose exploiting the information concerning the associated segments and the actual robot perceived pose. Consequently, the in visible variables are: `BackwardPosition`, the current perceived robot pose (the pose of the acquired segments before a Normalisation), and `AssociatedViews`, the pairs of associated segments.

When the `REGISTER` command is sent to the Performer, then the evaluation of the current robot position is executed. The estimated robot pose is made available by the `EstimatedPosition` out visible.

Likewise `Association`, this class is abstract. Our concrete implementation is the `MonteCarlo` class. The Registration is made by a randomatic selection of new positions that are quite near to the perceived one (as explained in subsection 6.3.3).

`Localisation` Performer is the core of the activity. It is in charge of executing a major number of activities activated by the following commands:

- `CREATE_LOCALISATION`. This command is executed at the end of every *Localisation* activity. The aim is to store the estimated located views (see chapter 3) in instances of `EstimatedLocatedView` class (7.3.3). Information needed are the current perceived view (from `PerceptionPerformer`) and the relative estimated pose of the robot (from the `Registration` Performer). These information are retrieved from the `PerceivedLocatedView` and `EstimatedPosition` in visible respectively. Special case of this command is `CREATE_FIRST-LOCALISATION`. This is used when the system is started up.

- `RELATIVE_LOCALISATION`. This command specifies to the Performer that a relative *Localisation* is required. A relative *Localisation* implies that the reference view is the previous view (i.e., the last estimated located view). At this aim, the Performer updates its `ReferenceView` out visible exploiting the last `EstimatedLocatedView` created (i.e., the result of the last execution of the `CREATE_LOCALISATION` command). Then, it updates also the `ViewToBackward` and `BackwardPosition` out visible variable to provide the `ViewBackwarder` Performer with the information it needs for Normalisation;

- GLOBAL_LOCALISATION. When an absolute *Localisation* is required, this command is sent to the Performer. The behaviour is similar to the RELATIVE_LOCALISATION command. The difference consists in the reference view it exports to the Association Performer: instead of being an estimated located view, this information is the world model.

- UPDATE_ABS-POS. Localisation Performer receives this command when the Registration phase has been executed. This command allows the Performer to estimate the new absolute robot pose. At this aim, it needs the current estimated position (readeable from its EstimatedPosition in visible). After the computation, it publishes it by means of its AbsolutePosition out visible;

- CREATE_ABSOLUTE-VIEW. The command specifies the Performer to produce the actual absolute view from the current perceived view. In particular, it reads from its AssociatedViews in visible the segments to rototranslate, and exploits the absolute robot pose estimated by the execution of the UPDATE_ABS-POS command. Then, the Performer exports the absolute view by means of its AbsoluteViews out visible.

7.4.4 Modelling activity

The *Modelling* activity is reified by the Performers sketched in figure 7.14.

When *Modelling* is required, the estimated located views from *Localisation* are made available to WMPerformer (see chapter 3). The command UPDATE_VIEWS allows the Performer to retrieve these views and to export them to the Fusion Performer. The other acceptable command of this Performer is the UPDATE_WM one: it is the command used to provide the LocalisationPerformer with an updated representation of the world model.

Fusion Performer aim is to merge all the homologous segments from the history of views. Since the history of views may be critical for the execution of the Fusion phase (see chapter 3), this Performer is a CriticalPerformer. When it receives the FUSE_VIEWS command, it verifies the validity of the views and then perform the fusion. If the views are invalid, then it notifies the criticality by means of its CriticalityNotifier out visible. The fusion is performed by exploiting the algorithm described in subsection 6.4.1. At the end of the activity, the Performer exports the fused segments (instance of the FusedSegments class) by means of its FusedSegments out visible.

Integration Performer aim is to update the world model by integrating the new information produced by the Fusion Performer. It accepts the INTEGRATE command that causes the execution of Integration phase as described in subsection 6.4.2. The updated world model is then given back to WMPerformer by means of the WorldModel out visible.

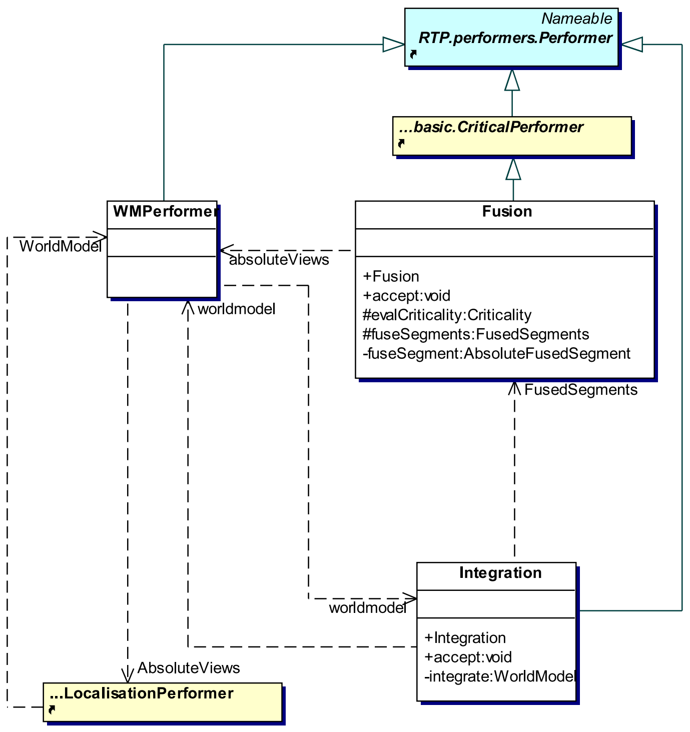


Figure 7.14: The Performers for *Modelling* purpose

7.5 System dynamics

This section deals with the dynamics of the CLAM system as described in chapter 3 and reified using RTP concepts. Subsection 7.5.3 describes the normal (concurrent) execution of the system, whereas subsection 7.5.4 will describe the management of criticalities. Before entering into dynamics details, some information about the startup of the system will be provided in the following subsection. The startup involves the topology creation, the RTP traces definition, and the clocks advancing time determination.

7.5.1 System Topology

The topology defines which out visibles are linked to which in visibles. As explained in section 5.3.3, RTP framework defines a `Topologist` Performer that is in charge of creating all the Performers and the Projectors constituting the system. Since Topologist is a Performer, it creates system components and connectors only upon command.

Commands are placed inside a trace enriched by their planned time intervals. An idea of the commands delivered to the Topologist may be found in the code fragment of figure 7.15. The code fragment sketches the commands for the creation of the `ViewAcquisition` Performer, the `PerceptionPerformer`, and the Projector linking the `PerceivedView` out visible exported by the `AcquisitionView` Performer and the `PerceivedView` in visible of the `PerceptionPerformer`. Similar commands complete the overall topology.

When all commands are added to the trace, then they may be delivered to `Topologist` Performer.

7.5.2 System Configuration for Concurrence

The configuration of the system involves the definition of traces, virtual clocks, and reference clocks. Their definitions are imposed by the following considerations:

1. *Localisation* and *Modelling* activities are driven by different timings since they produce information characterised by different lifetime (see section 3.5)). For this reason, commands performing *Localisation* and *Modelling* should be placed inside two different RTP TimedTraces each of them driven by its own RTP virtual clock (see subsection 5.4.2) with proper period.

2. time scale of the *Localisation* virtual clock is finer then the *Modelling* time scale, due to the short lifetime of *Localisation* produced information;

```
//...
//The creation of the ViewAcquisition Performer
trace.addRequest(new TimedRequest(new TimeInterval(begin, begin+=step),
   new CreatePerformerCommand("clam.performers.perception.ViewAcquisition",
   new GenericName("ViewAcquisition")),
   new GenericName("Topologist")));
//The creation of the PerceptionPerformer
trace.addRequest(new TimedRequest(new TimeInterval(begin, begin+=step),
   new CreatePerformerCommand("clam.performers.perception.PerceptionPerformer",
   new GenericName("PerceptionPerformer")),
   new GenericName("Topologist")));
//The creation of the Projector between the PerceivedView out visible of
//ViewAcquisition and the PerceivedView in visible of PerceptionPerformer
trace.addRequest(new TimedRequest(new TimeInterval(begin, begin+=step),
   new CreateProjectorCommand(
   new GenericName("ViewAcquisition"), new GenericName("PerceptionPerformer"),
   new GenericName("PerceivedView"), new GenericName("PerceivedView"),
   new GenericName("ViewAcquisition_PerceptionPerformer")),
   new GenericName("Topologist")));
//...
```

Figure 7.15: A fragment of topology creation

3. the execution timings of *Localisation* and *Modelling* are independent since they operates on separate information. Consequently, the time advancement of the two virtual clocks is controlled by two different reference clocks.

In term of concrete implementation, the Strategist is in charge of defining virtual clocks, traces, and reference clocks. Successively, Strategist fills the traces and finally starts the system by activating the reference clocks.

Referring to code fragment 7.16, the Strategist creates two virtual clocks (comments 1 and 4). `vcLocalisation` starts at time 0, ends at time 1,000,000,000, has a now value equals to 0, and its time scale 1. This means that its now value will be incremented each time the reference clock controlling its time advancement will tick it. `vcModelling` has the same parameters values of `vcLocalisation` except for the time scale: 16 means that `vcModelling` now value will be incremented at every 16 ticks of the reference clock controlling its time advancement.

The Strategist creates two separated `TimedTraces`: one will contain commands that will be delivered to *Localisation* Performers (comments 2 and 3), the second commands to *Modelling* Performers (comments 5 and 6). *Localisation* timed trace keeps track of its current time by means of `vcLocalisation` virtual clock (comment 2), whereas *Modelling* timed trace exploits `vcModelling` virtual clock (comment 5).

```
VirtualClock vcLocalisation = new VirtualClock(0, 1000000000, 0, 1); //1
TimedTrace t1 = new TimedTrace(vcLocalisation);                     //2
addTrace(t1, new GenericName("Localisation"));                      //3

VirtualClock vcModelling = new VirtualClock(0, 1000000000, 0, 16); //4
TimedTrace t2 = new TimedTrace(vcModelling);                        //5
addTrace(t2, new GenericName("Modelling"));                         //6
```

Figure 7.16: The TimedTraces creation

The Strategist creates two different Engines that are in charge of delivering commands to the Performers (comments 1 and 3 in code fragment 7.17). `engLocalisation` dispatches commands placed in the *Localisation* TimedTrace (comment 2), whereas `engModelling` is in charge of delivering commands placed in the *Modelling* TimedTrace (comment 4).

```
TickedEngine engLocalisation = new TickedEngine(); //1
engLocalisation.addTrace(t1);                      //2

TickedEngine engModelling = new TickedEngine(); //3
engModelling.addTrace(t2);                         //4
```

Figure 7.17: The Engines creation

Finally, the Strategist creates two reference clocks, one for each virtual clock, (see code fragment 7.18).

```
ReferenceClock rcLocalisation = new ReferenceClock(); //1
rcLocalisation.addEngine(engLocalisation);            //2
rcLocalisation.addVC(vcLocalisation);                 //3
rcLocalisation.addStrategist(this);                   //4

ReferenceClock rcModelling = new ReferenceClock(); //5
rcModelling.addEngine(engModelling);                  //6
rcModelling.addVC(vcModelling);                       //7
rcModelling.addStrategist(this);                      //8
```

Figure 7.18: The Reference clocks creation

Before starting the system, i.e., starting the reference clocks, the Strategist fills the TimedTraces with initial requests. Referring to code fragment 7.19:

- comment 1: the `setUpRequest` method builds the requests for topology creation. The requests are inserted in *Localisation* TimedTrace[1] and the planned begin for the first request is 0;
- comment 2: the `perception` method builds the requests for a *Perception* activity. The requests are inserted into the *Localisation* TimedTrace. The planned begin for the first *Perception* request is the planned end of the last request for topology creation (`endPlannedSetup`);
- comment 3: the `localisation` method builds the request for a *Localisation* activity. The requests are inserted into the *Localisation* TimedTrace. The planned begin for the first *Localisation* request is the planned end of the last request for *Perception* activity (`endPlannedPerception`);
- comment 4: the `modelling` method builds the requests for a *Modelling* activity. The requests are inserted into the *Modelling* TimedTrace. The planned begin for the first *Modelling* request is the planned end of the last request for topology creation (`endPlannedSetup`).

```
long endPlannedSetup = setUpRequests(t1, 0, 2);          //1
long endPlannedPerception = perception(t1, endPlannedSetup); //2
localisation(t1, endPlannedPerception);                  //3
modelling(t2, endPlannedSetup);                         //4
```

Figure 7.19: The planning of one *Perception*, one *Localisation* and one *Modelling* activities

Figure 7.20 sketches the system state after the described configuration[2]. The virtual clock of *Localisation* TimedTrace is finer with respect to *Modelling* one and the future traces contains the requests for the first *Localisation* and the first *Modelling*. The figure shows also how the timings are independent and that *Localisation* and *Modelling* are concurrent.

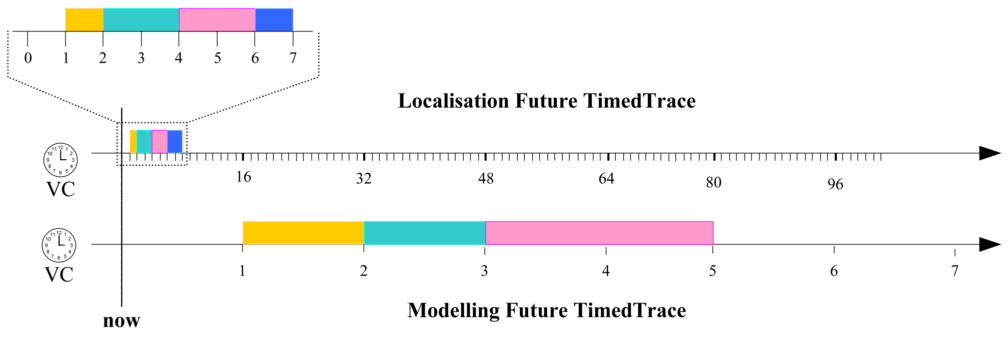
7.5.3 Concurrence Management

During normal condition (i.e., when criticalities do not arise, see section 3.3), *Localisation* and *Modelling* are executed concurrently, i.e., each activity is carried on independent from the other.

The two main activities and the *Perception* one are subdivided into phases (see chapter 6) each of them may be activated if the corresponding request (or set of requests) is present inside the related TimedTrace.

[1]It is possible to place requests in the *Modelling* trace too.

[2]Actually, in the *Localisation* TimedTrace are also inserted the requests for topology creation and the first *Perception* activity.



🟨 Normalisation /Fusion
🟦 Association/ Integration
🟪 Registration / WM
🟦 Localisation

Figure 7.20: The system state after the configuration

Requests for *Perception* and *Localisation* are placed inside the same timed traced.

When one command (inside the request) is delivered to the Performer depends on both the now value of the TimedTrace and the planned interval of the request. It is in charge of the Strategist to put the requests inside the traces with a right planned interval. The order in which commands must be delivered is given by the object flows and the transitions present in the activity diagram of figure 3.5. In the same diagram, it is enphasised that a *Localisation* follows immediately one *Perception* (the dashed arrow connecting pv:PV to Normalisation). For this reason, the request for *Perception* are placed inside the TimedTrace we use for *Localisation*.

From the above consideration and referring to figure 3.5, the Strategist must compute the planned intervals of the requests driving *Localisation* so that the order in which the requests are selected is like the one shows in table 7.1:

- the first operation to perform is to acquire the current perceived robot pose. Performer involved is the `PositionAcquisition` one that is activated by means of the `ACQUIRE_POSITION` command (row numbered 1);

- the images from cameras are then acquired and elaborated to produce the current perceived view. Performer involved is the `ViewAcquisition` one and the relative command is the `ACQUIRE_VIEW` (row numbered 2);

- from the acquired perceived view (row numbered 3) and the acquired perceived pose (row numbered 4), then the PerceptionPerformer merges the perceptive inputs (row numbered 5) to produce current PerceivedLocatedView. *Perception* activity is so completed;

- LocalisationPerformer acquires the actual PerceivedLocatedView (row numbered 6) and exports to the interested Performers the reference view (the last estimated view), the pose vector to which refer current perceived view, and the current acquired view (row numbered 7);
- ViewBackwarder Performer refers current perceived view (row numbered 8) to the coordinate system (row numbered 9) given by the reference view (row numbered 10);
- Association Performer associates (row numbered 13) segments from the reference view (row numbered 12) and the actual perceived view referred to the same coordinate system of the reference view (row numbered 11)
- Registration Performer estimates the actual robot position (row numbered 16) exploiting the set of associated segment (row numbered 14) and the actual perceived pose (row numbered 15);
- Localisation Performer then creates the new EstimatedLocatedView (row numbered 18) exploiting the estimated pose (row numbered 17), it updates the absolute estimated robot pose (row numbered 119), and then it creates the new absolute view (row numbered 21) by means of the associated views (row numbered 20).

The information exchange between Performers is made by means of dedicated Projectors (see row numbered 3, 4, 6, 8, 9, 11, 12, 14, 15, 17, and 20). The only command they recognise is the SYNC one. When they receive the command, they read from the out visible of the Performer exporting the information and write to the corresponding in visible of the Performer requiring the information.

Figure 7.21 shows a screen shot of the application we developed. The graphical interface is divided into two main panel. The top panel shows activities concerning *Localisation*, the bottom the *Modelling*. In the top panel:

- on the left side is presented the current perceived view;
- on the left bottom side is presented the reference view;
- on the right side is presented the current perceived view rototranslated with respect to the reference view;
- on the right bottom side is presented the associated segments;

	Command	**Recipient**
1	ACQUIRE_POSITION	PositionAcquisition
2	ACQUIRE_VIEW	ViewAcquisition
3	SYNC	ViewAcquisition_PerceptionPerformer
4	SYNC	PositionAcquisition_PerceptionPerformer
5	PERCEPTION	PerceptionPerformer
6	SYNC	PerceptionPerformer_LocalisationPerformer
7	RELATIVE_LOCALISATION	LocalisationPerformer
8	SYNC	LocalisationPerformer_ViewBackwarder.View
9	SYNC	LocalisationPerformer_ViewBackwarder.Position
10	BACKWARD	ViewBackwarder
11	SYNC	ViewBackwarder_Association
12	SYNC	LocalisationPerformer_Association
13	ASSOCIATE	Association
14	SYNC	Association_Registration
15	SYNC	LocalisationPerformer_Registration
16	REGISTER	Registration
17	SYNC	Registration_LocalisationPerformer
18	CREATE_LOCALISATION	LocalisationPerformer
19	UPDATE_ABS-POS	LocalisationPerformer
20	SYNC	Association_LocalisationPerformer.AssociatedViews
21	CREATE_ABSOLUTE-VIEW	LocalisationPerformer

Table 7.1: Commands and Recipients for *Perception* and *Localisation*

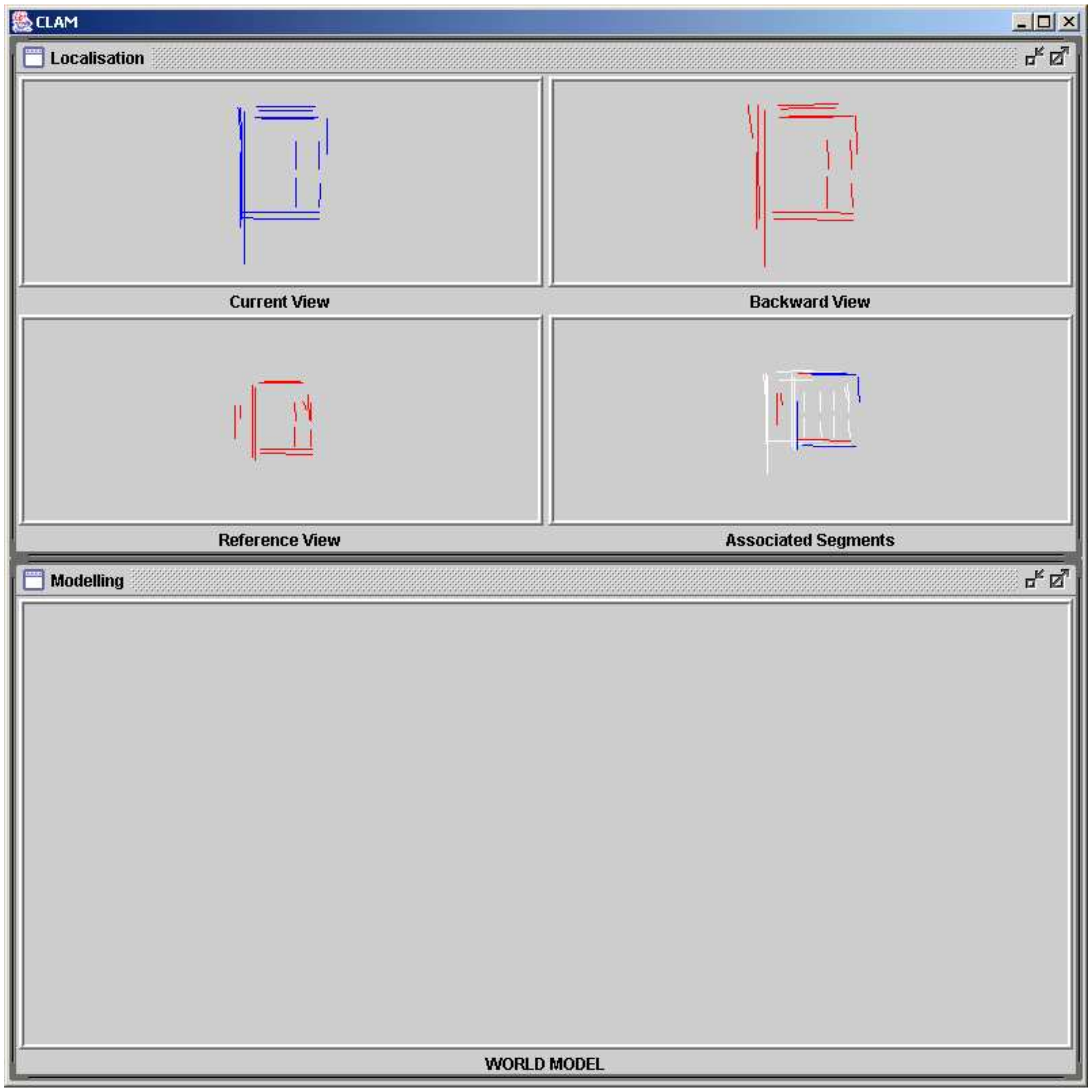


Figure 7.21: The execution of *Localisation* step

	Command	**Recipient**
1	SYNC	LocalisationPerformer_WMPerformer
2	UPDATE_VIEWS	WMPerformer
3	SYNC	WMPerformer_Fusion
4	FUSE_VIEWS	Fusion
5	SYNC	Fusion_Integration
6	SYNC	WMPerformer_Integration
7	INTEGRATE	Integration
8	SYNC	Integration_WMPerformer
9	SYNC	WMPerformer_LocalisationPerformer

Table 7.2: Commands and Recipients for *Modelling*

The figure does not show the world model in the bottom panel: this means that *Modelling* activity is not yet completed.

Since timings are independent, the requests planned interval driving *Modelling* are completely unrelated to the ones driving *Localisation*. Likewise *Localisation*, the Strategist must determinate the planned interval so that the associated commands are delivered in the order in which are presented in table 7.2.

Referring to table 7.2, the *Modelling* execution sequence is the following:

- the first operation is the creation of the history of views that will be exploited by the `Fusion` performer (row numbered 2);
- Fusion Performer fuses homologous segments (row numbered 4) exploiting the history of views (row numbered 3);
- Integration Performer updates the world model (row numbered 7) exploiting the fused segments (row numbered 5) and the actual world model (row numbered 8). Finally, it returns an updated version of the world model to the `WMPerformer` (row numbered 9).

The same we said for information exchange between Performers in *Localisation* activity, still holds for *Modelling*.

Figure 7.22 sketches the situation of the system in which at least one Modelling step has been executed.

During system evolution, the Strategist observes the overall behaviour and, when necessary, updates the TimedTraces with new sets of requests as described in tables 7.1 and 7.2. The necessity arises when the requests are almost all delivered (successfully).

Concerning *Localisation*, code fragment 7.23 shows the policy adopted by the Strategist: when the last executed request is `CREATE_ABSOLUTE-VIEW` (comment 4), then *Localisation* process is nearly finished (see table 7.1).

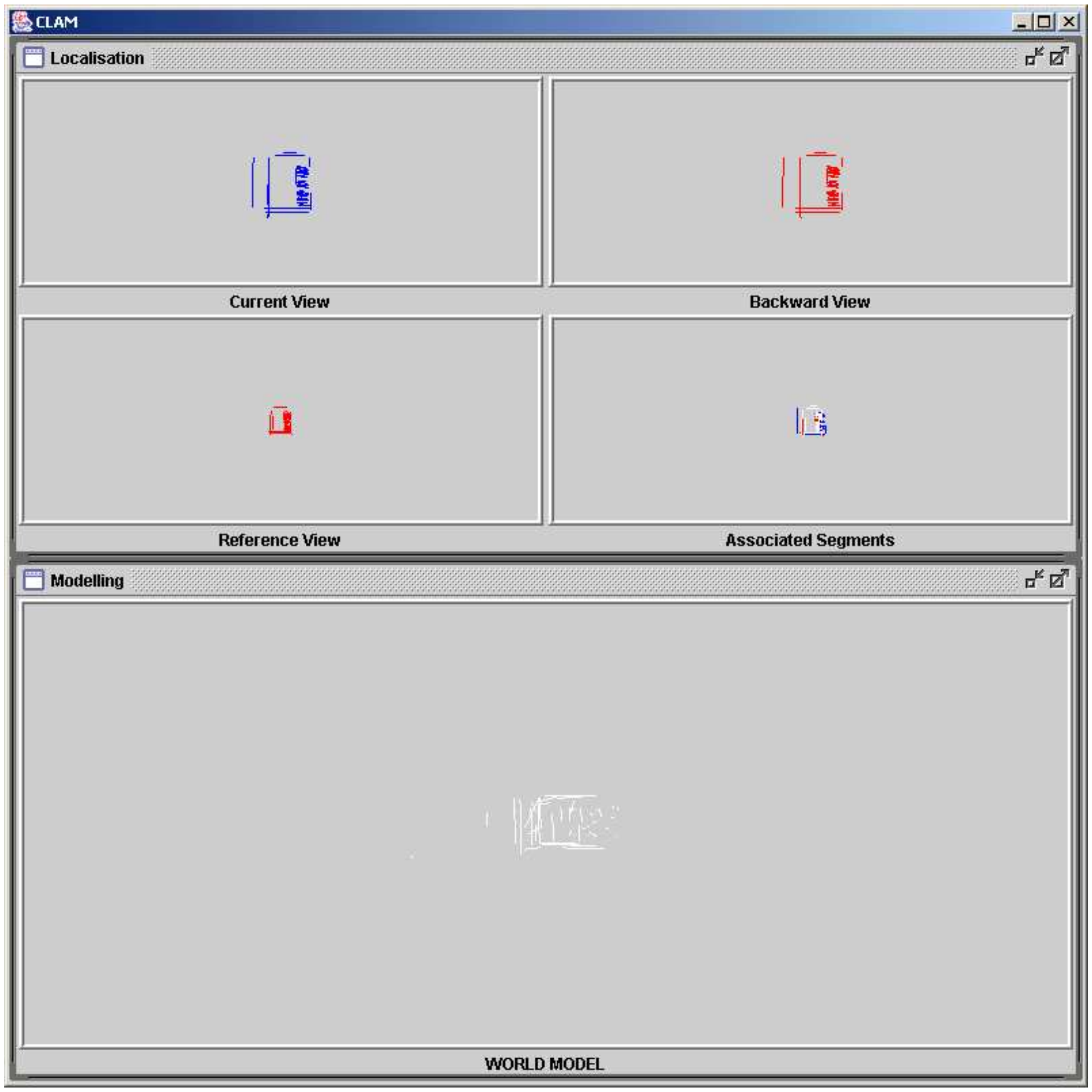


Figure 7.22: The system at runtime

Since the Strategist observes the system each time the reference clocks advances its time, it may happen that the last executed request is still the same (the time advancement of the *Localisation* virtual clock may not be equals to its reference clock). To overcame this situation, the Strategist, before updating the timed trace verifies also that there is not already planned another *Perception-Localisation* activity (comment 3). If all the two conditions hold, then it plans a new set of requests for *Perception* and then *Localisation*. The first request begin planned time is set equals to the planned end of the last request to perform present in the TimedTrace (comment 6). If there is no request inside the trace, then the time is set equals to the now value of the *Localisation* TimedTrace virtual clock. The others begin planned time are set consequently.

```
Request requestLocLastDone = t1.lastDone();                    //1
Request alreadyInserted = t1.searchFor(
   new GenericCommand("BACKWARD"), new GenericName("ViewBackwarder")); //2
if(requestLocLastDone != null && alreadyInserted == null) {     //3
    if(requestLocLastDone.getCommand() instanceof GenericCommand){
      GenericCommand c = (GenericCommand)requestLocLastDone.getCommand();
      if(c.getCommand().equals("CREATE_ABSOLUTE-VIEW")){     //4
         TimedRequest lastToDo = (TimedRequest)t1.lastTodo(); //5
            long durationLocalisation = 0;
         if(lastToDo != null) {
            durationLocalisation = lastToDo.getPlanned().getEnd(); //6
            } else {
            durationLocalisation = vcLocalisation.now();        //7
            }
         durationLocalisation = perception(t1, durationLocalisation); //8
         durationLocalisation = localisation(t1, durationLocalisation); //9
      }
    }
}
```

Figure 7.23: The planning of new Requests under concurrent execution

The technique the Strategist uses for *Localisation* still applies to *Modelling*. Referring to code fragment 7.24, the Strategist controls if the last performed command was `INTEGRATE` (comment 4) and that it did not planned another *Modelling* activity (comment 3). The planned begin time for the first new request is set equals to the planned end of the last request presents in the future trace, or equals to the now value of the *Modelling* TimedTrace virtual clock.

Figure 7.25 sketches a graphical representation od the concurrence of *Localisation* and *Modelling* activities. They are carried on with independent timings over two TimedTraces (with different period). The Strategist

```
Request requestModLastDone = (TimedRequest)tMod.lastDone();  //1
alreadyInserted = t2.searchFor(
   new GenericCommand("UPDATE_VIEWS"), new GenericName("WMPerformer")); //2
if(requestModLastDone != null && alreadyInserted == null ){   //3
    if(requestModLastDone.getCommand() instanceof GenericCommand){
      GenericCommand cM = (GenericCommand)requestModLastDone.getCommand();
      if(cM.getCommand().equals("INTEGRATE")){              //4
         TimedRequest lastToDo = (TimedRequest)t2.lastTodo(); //5
         long durationModelling = 0;
         if(lastToDo != null) {
            durationModelling = lastToDo.getPlanned().getEnd();   //6
            } else {
            durationModelling = vcModelling.now();             //7
            }
         modelling(t2, durationModelling);                     //8
      }
    }
}
```

Figure 7.24: The planning of new Requests under concurrent execution

observes the past TimedTraces of both the activities, and plans future behaviour.

7.5.4 Criticalities Management

When criticalities arise, then the Strategist must opportunely synchronise the two activities. The Strategist becomes aware of criticalities since it is notified by the `CriticalPerformers` (by means of their `CriticalitiesNotifier` out visibles).

Referring to section 3.4 of the chapter describing the CLAM approach, criticalities occur when:

- the reference view is not a plausible representation of the environment at the moment in which actual perceived view was captured;
- the history of views do not contain information suitable for the update of the world model.

First pointed criticality concerns *Localisation*, whereas the second one, *Modelling*.

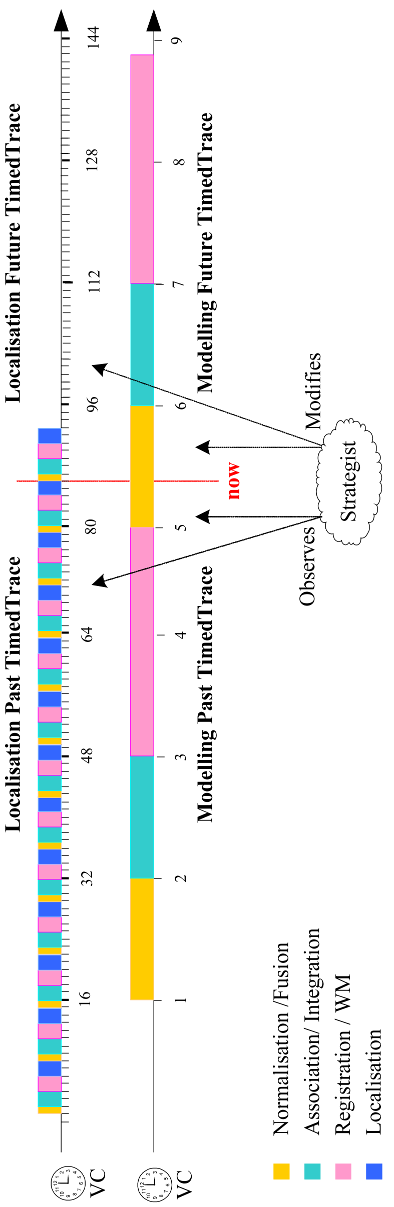


Figure 7.25: The RTP management of concurrent execution

	Command	**Recipient**
1	UPDATE_WM	WMPerformer
2	SYNC	WMPerformer_LocalisationPerformer
3	SYNC	Association_LocalisationPerformer.BackwardView
4	GLOBAL_LOCALISATION	LocalisationPerformer
5	SYNC	LocalisationPerformer_ViewBackwarder.View
6	SYNC	LocalisationPerformer_ViewBackwarder.Position
7	BACKWARD	ViewBackwarder
8	SYNC	ViewBackwarder_Association
9	SYNC	LocalisationPerformer_Association
10	ASSOCIATE	Association

Table 7.3: Commands and Recipients for the criticality management

7.5.4.1 Localisation activity

Since the plausibility of the reference view is verified in the Association phase (3.4), then criticalities may only arise from the execution of the `ASSOCIATE` command delivered to the `Association` Performer. As a matter of the fact, the `Association` Performer has been defined a `CriticalPerformer` (see subsection 3.4.1).

When criticalities arise, the `Association` Performer notifies the Strategist by writing in its notifying out visible the criticality typology: `NoAssociatedSegments` (i.e., no segment of actual perceived view has been found in the reference view) or `FewAssociatedSegments` (i.e., a few number of associeted segments has been found).

Apart from the specific type of criticality, this may be overcame by substituting current reference view with the world model and then going on with normal execution.

The substitution of the world model is obtained by the requests as described in table 7.3:

- the first request commands the `WMPerformer` to export the current world model (row numbered 1);
- the next request synchronises *Localisation* and *Modelling* activities (row numbered 2). The request is delivered to the Projector linking the world model out visible of the `WMPerformer` to the world model in visible of the `LocalisationPerformer`;
- the `LocalisationPerformer` exports the current perceived view, the world model, and the pose vector to which refer current perceived view (row numbered 4);
- `ViewBackwarder` Performer refers current perceived view (row numbered 5) to the coordinate system (row numbered 6) given by the

reference view (row numbered 7);

- `Association` Performer associates (row numbered 10) segments from the reference view (row numbered 9) and the actual perceived view referred to the same coordinate system of the reference view (row numbered 8)

Then, the system may go on with the requests that already were in the timed traces. Obviously their planned time intervals must be changed ro reflect the new added requests, i.e., they must be deferred.

The above strategy is described in code fragment 7.26 (comments 1, 2, and 3). First the Strategy plans the requests described in table 7.3 (comment 2), then it defers the requests presents in the timed trace (comment 3). These requests are those presents in table 7.1 from the row numbered 14 to the end.

```
if(criticality instanceof FewAssociatedSegments ||
   criticality instanceof NoAssociatedSegments){          //1
   long time = globalLocalisation(t1, vcLocalisation.now()); //2
   adjustLocalisationFutureTrace(t1, time);               //3
} else {
   if(criticality instanceof NoWorldModel){               //4
      vcModelling.speedUp();                              //5
      vcLocalisation.slowDown();                          //6
      long time = globalLocalisation(t1, vcLocalisation.now()); //7
      adjustLocalisationFutureTrace(t1, time);            //8
   }
}
```

Figure 7.26: The management of *Localisation* criticalities

Figure 7.27 shows a screen shot of our implementation. In the panel containing the reference view is visible the world model. This means that the criticality has been correctly managed by the Strategist.

If the world model does not still exist (comment 4), then the Strategist speeds up the *Modelling* activity (comment 5), slows down the *Localisation* one (comment 6), and adopts the same strategy used above (comments 7 and 8).

Figure 7.28 sketches the management of the above *Localisation* criticality. When a criticality occurs (figure 7.28.a), the Strategist adjusts the relative speeds of both *Localisation* and *Modelling* activities to allows synchronisation (figure 7.28.c). It slows down the *Localisation* virtual clock, speeds up the *Modelling* virtual clock, and modifies the *Localisation* Timed-Trace (figure 7.28.b).

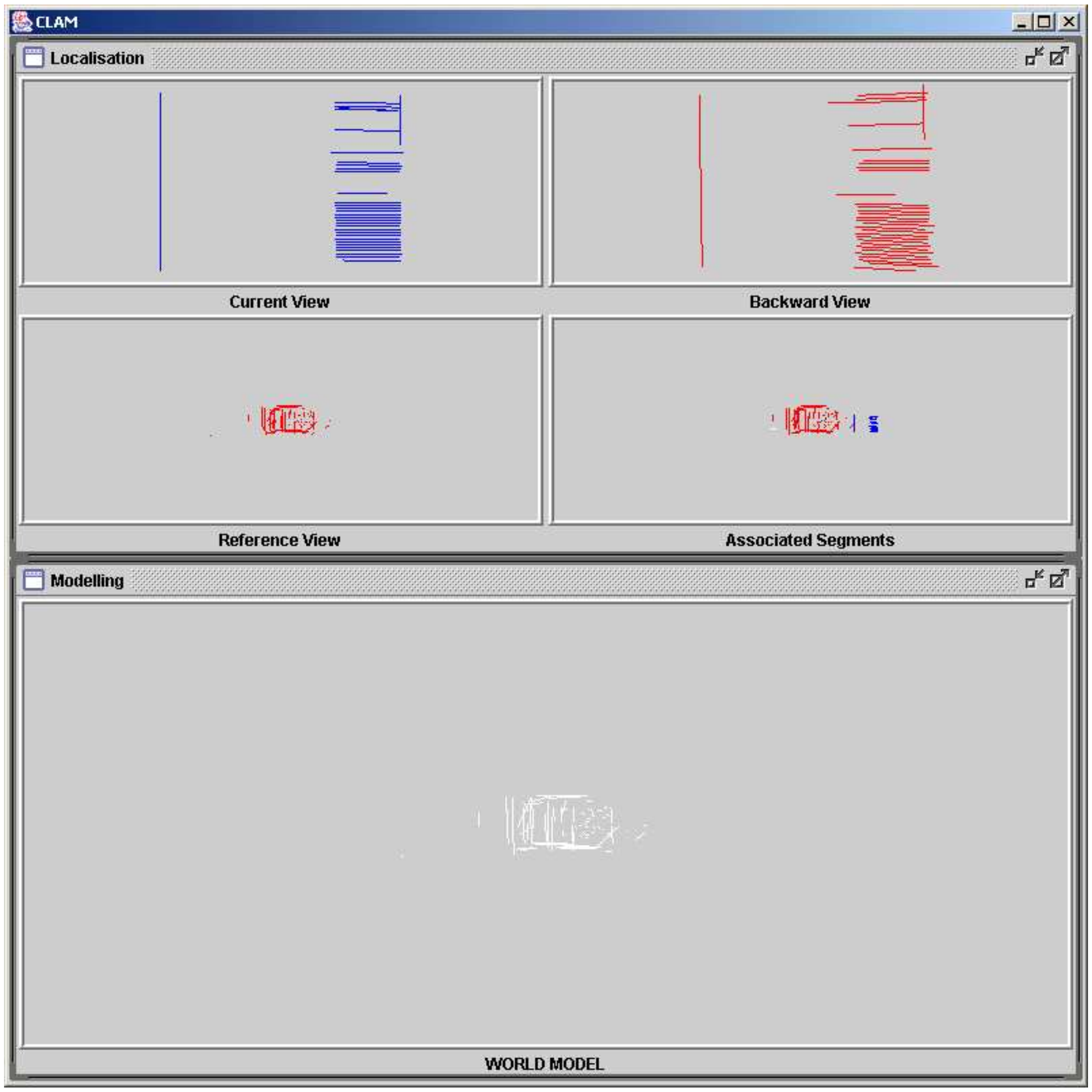


Figure 7.27: A criticality concerning *Localisation*

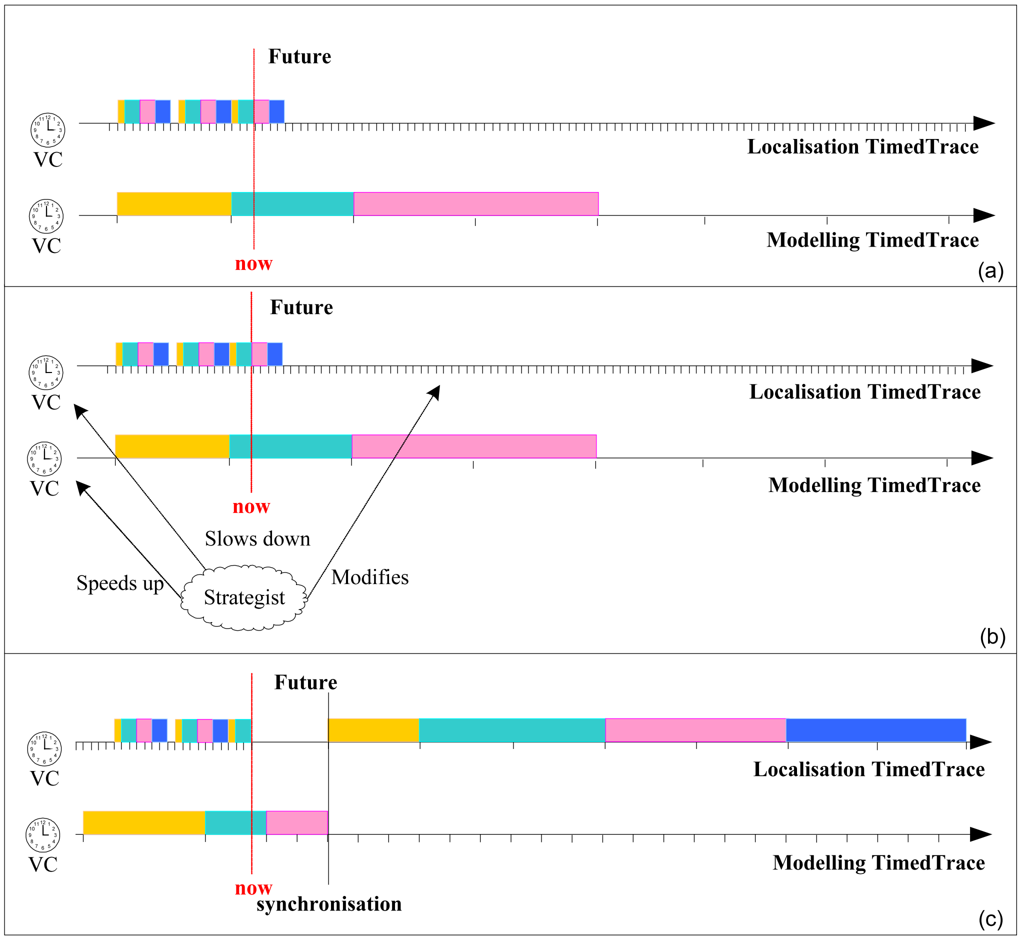


Figure 7.28: The RTP management of *Localisation* criticality

If the criticalities still overcame, then other strategies must be adopted such as the planning a different robot speed or a different robot direction. This is out of our scope.

7.5.4.2 Modelling activity

Since the plausibility of the history of views is verified in the Fusion phase, then criticalities may only arise from the execution of the `FUSION` command delivered to the `Fusion` Performer (likewise `Association` Performer, `Fusion` is a `CriticalPerformer`).

When criticalities arise, the `Fusion` Performer notifies the Strategist by writing in its notifying out visible the criticality.

This criticality may solved by adjusting the relative speeds of the activities (see subsection 3.4.2).

Referring to code fragment 7.29, the Strategist first slows down the virtual clock of *Modelling* activity (comment 2), then speeds up the virtual clock of *Localisation* activity (comment 3), and finally adjusts the TimedTrace of *Modelling*. The adjustment involves the insertion of requests from row numbered 1 to row numbered 4 in table 7.2 (comment 4), and the deferment of the requests from row numbered 5 to row numbered 9 in table 7.2 (comment 5).

```
if(criticality instanceof NoAbsoluteViewsAvailable){    //1
   vcModelling.slowDown();                               //2
   vcLocalisation.speedUp();                             //3
   long time = repeatModelling(t2, vcLocalisation.now()); //4
   adjustModellingFutureTrace(t2, time);                 //5
}
```

Figure 7.29: The management of *Modelling* criticality

Figure 7.30 sketches the management of the above *Modelling* criticality. When a criticality occurs (figure 7.30.a), the Strategist adjusts the relative speeds of both *Localisation* and *Modelling* activities to allows synchronisation (figure 7.30.c). It slows down the *Modelling* virtual clock, speeds up the *Localisation* virtual clock, and modifies the *Modelling* TimedTrace (figure 7.30.b).

If the criticality still arises then other strategies must be adopted as explained in the previous subsection.

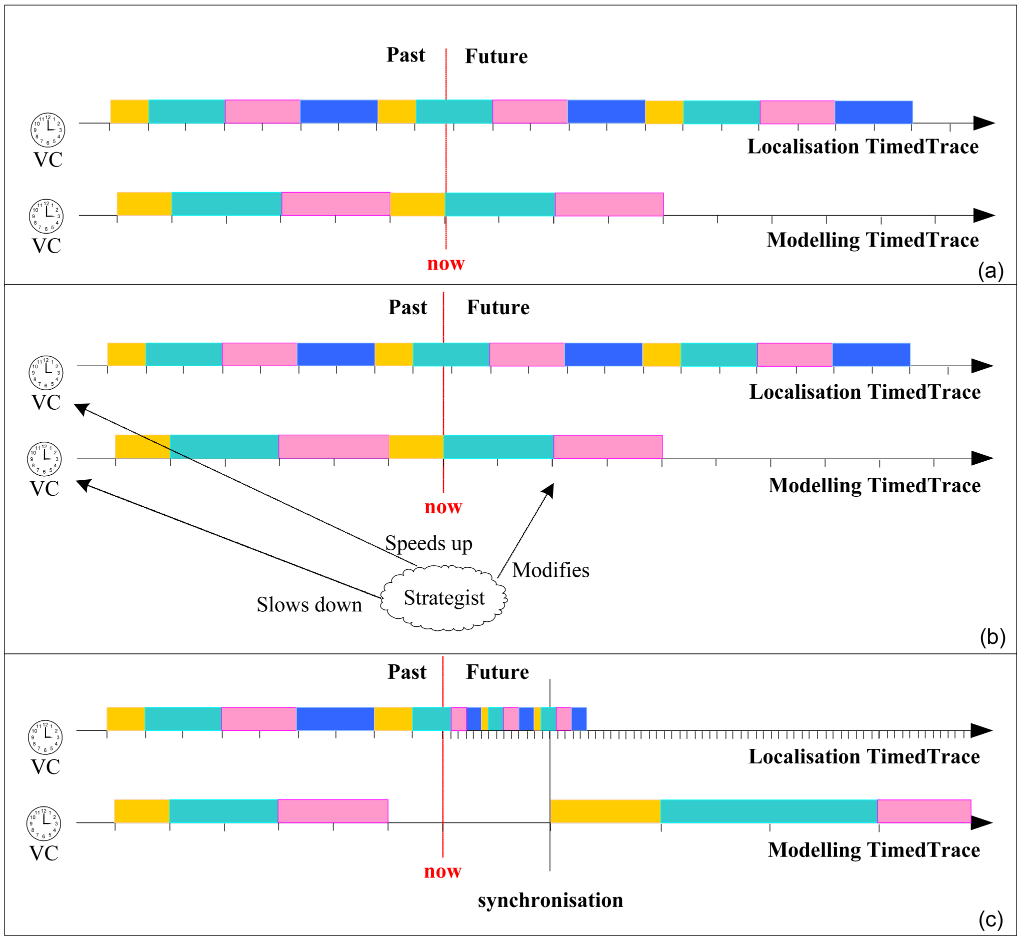


Figure 7.30: The RTP management of *Modelling* criticality

Chapter 8

Conclusions

8.1 Summary of Contribution

The thesis has presented an interdisciplinary work involving the area of autonomous mobile robotics and the field of software architecture.

Concerning the robotics area, we have proposed a novel approach to solve the problem of robot exploration in an unknown environment and with uncertain information about its pose. The problem has been addressed with an architectural approach. This approach has led us to the definition of CLAM (Concurrent Localisation And Mapping).

CLAM is based on assumption that, in normal condition (i.e., when the information an activity relies on is reliable), the two main activities (*Localisation* and *Modelling*) that a robot performs may be executed concurrently. When some kind of synchronisation is needed to accomplish the reliability of the information, then it may be realised by modifying opportunely the relative speed of the activities. Formally, the key concepts of the CLAM approach are the following:

1. *Localisation* and *Modelling*, both relying on *Perception*, are the basic activities performed by a robot when exploring an unknown environment;

2. *Localisation* and *Modelling* operate on *separate information* and are subject to *different timing constraints*. Therefore they can be performed *concurrently* and with *independent timings*;

3. Localisation relies on information which loosely depends on the information generated by *Modelling*, and vice-versa. Therefore *Localisation* and *Modelling* must *synchronise* whenever a *criticality* arises, i.e., whenever the information an activity relies on is not reliable;

4. Synchronisation is controlled by a *strategy* which relies on the observation of the criticalities and drives the relative rates of the activities.

The CLAM approach is based on a proper separation of concern between *Localisation* and *Modelling* to break the chicken-and-egg loop. The activities, even if each other related, may be considered independent each other since they operate with independent timings. When criticalities arise, then a suitable strategy must drive opportunely their execution rates so that the syncronisation is successfully reached.

A concrete CLAM implementation needs an underlying software architecture capable of capturing the temporal aspects belonging to the system itself. From CLAM key concepts, it follows that a CLAM system needs to execute different activities with different, dynamic, and inter-dependent temporal requirements. Moreover a CLAM system needs to dynamically change the activities temporal requirements. From the above consideration, a CLAM system may be considered as a time-sensitive one.

Real-Time Performers is an architectural framework based on reflection that allows monitoring and control of the time related behavioural aspects of the systems built upon it. RTP is based on the following concepts:

- the system runs *temporally planned actions*;
- actions are planned for execution by placing them in *timelines* (defining the overall behaviour of the system);
- a timeline is “ticked” by a *virtual clock*;
- a virtual clock may be speed-tuned to modify the execution rates of actions on its timeline;
- a *strategist* may control the system by tuning virtual clocks an by changing the content of timelines (i.e. adding/removing/modifying actions).

The RTP framework was implemented using the Java Programming Language. The RTP framework was used to build a concrete implementation of a system based on CLAM principles.

Preliminary qualitative testing has been done to verify that the management of CLAM criticalities works as expected. The implemented system actually succeeds in changing strategy and timings when needed.

8.1.1 Publications

The results of this thesis have been published in the following referred paper:

- D. Micucci, M. Sarini, C. Simone, F. Tisato, and A. Trentini. *Conceptual and concrete architectures in the design of CSCW application.* In Proceedings of the annual Workshop on Agents. November, 2002, Milan

- D. Micucci, A. Trentini, and F. Tisato. *A connector-based approach for controlled data distribution RTP architecture.* In Proceedings of the Data Distribution for Real-Time Systems. May, 2003 Providence, RI, USA
- D. Micucci and A. Trentini. *A pattern-like framework to ease dynamically change components behaviour.* In Proceedings of the 15th international conference on Software Engineering and Knowledge Engineering. July, 2003, Redwood City, CA, USA
- D. Micucci. *An Object-Oriented software approach for a distributed human tracking motion system.* In Proceedings of Visual Communications and Image Processing (VCIP). SPIE. July, 2003, Lugano, Switzerland

The following papers are under revision:

- D. Micucci, S. Ruocco, F. Tisato, A. Trentini. *Time Sensitive Architectures: a Reflective Approach.* Submitted to SAC 2004
- D. Micucci, F. Marchese, D. Sorrenti, F. Tisato. *CLAM: Concurrent Localisation And Mapping from an architectural point of view.* To submit to IROS 2004, IEEE/RSJ International Conference on Intelligent Robots and Systems

8.2 Future Developments

Our first concern is a thorough validation of the whole architecture. We are setting up qualitative and quantitative test sets to measure:

- localisation error;
- model quality (shape, dimensions, etc.);
- correctness of criticality identification;
- RTP time constants correctness (and their tuning) with respect to the robot-world constants;
- resonances/loops;

Currently, due to problematic availability of the actual robot, every test of our system is done in batch, offline with respect to the robot itself: we capture sensors data and feed them into CLAM. CLAM then works on robot movements in a "stored" reality. Our next step is the complete integration into the robot guidance system.

The whole architecture is completely independent from the algorithms used for *Localisation* and *Modelling*. We would like to experiment with new algorithms (e.g. 3D shape recognition) to let new criticalities arise and try to manage them. The same pattern applies for new world-modelling algorithms (e.g. "segment cleaning").

Our current environment is somewhat static, instead we would like to experiment with a very dynamic one, and see if this approach compares well with other, more traditional, ones.

Moreover, to further test the validity of Real-Time Perfomers ideas, the framework is being exploited in a complex distributed webcam-based movement tracking system.

Bibliography

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedins of the International Conference on Software Engineering*, May 1994.

[2] J. Anders. Free MPEG Java Player, 2003. `http://rnvs.informatik.tu-chemnitz.de/~jan/MPEG/MPEG_Play.html`.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language (3_th edition).* Addison-Wesley, 2000.

[4] K. O. Arras and S. J. Vestli. Hybrid, high-precision localization for the mail distributing mobile robot system mops. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 1998.

[5] K. S. Arun, T. S. Huang, and S. D. S. D. Blostein. *IEEE Trans. Pattern Anal. Mach. Intelligence*, volume 9, chapter Least squares fitting of two 3-D point sets, pages 698–700. 1987.

[6] N. Ayache. *Artificial Vision for Mobile Robots: Stereo Vision and Multisensory Perception.* The MIT Press, 1991.

[7] N. Ayache and O.D. Faugeras. HYPER : A new Approach for the Recognition and Positioning of TwoDimensional Objects. In *IEEE Trans. on Patern Analysis and Machine Intelligence*, volume 8(1), pages 44–54, 1986.

[8] P.J. Besl and H.D. McKay. A method for registration of 3-d shapes. *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 14(2):239–256, February 1992.

[9] G. Blais and M. D. Levine. *IEEE Trans. Pattern Anal. Mach. Intelligence*, chapter Registering multiview range data to create 3D computer objects, pages 820–824. 1995.

[10] J. Borenstein, E. Everett, and F. Feng. *Navigating Mobile Robots: Systems and Techniques.* A. K. Peters, Ltd., Wellesley, MA, 1996.

[11] W. Burgard, A.B. Cremers, D. Fox, D. Hahnel, D. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. In *Artificial Intelligence*, volume 114(1-2), pages 3–55, 1999.

[12] W. Burgard, D. Fox, D. Hennig, and T. Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1996.

[13] W. Burgard, D. Fox, H. Jans, C. Matenar, and S. Thrun. Sonar-based mapping of large-scale mobile robot environments using em. In *Proceedings of the International Conference on Machine Learning*, 1999.

[14] J. A. Castellanos and J. D. Tardos. Mobile Robot Localisation and Map Building: a multisensor fusion approach. In *Kluwer Academic Publisher*, 1999.

[15] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering and 6th Reengineering Forum*, March 1998.

[16] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. A fresh look at programming-in-the-large. In *Proceedings of The Twenty-Second Annual International Computer Software and Application Conference (COMPSAC 98)*, August 1998.

[17] R. Chatila and J.-P. Laumond. Position referencing and consistent world modeling for mobile robots. In *Proceedings 1985 IEEE International Conference Robotic Automation*, volume 5(4), pages 138–145, 1985.

[18] H. Choset, I. Konuksven, and A. Rizzi. Sensor Based Planning: A Control Law for Generating the Generalized Voronoi Graph. In *Proceedings IEEE Int. Advanced Robotics*, 1996.

[19] I. J. Cox. *IEEE Transactions on Robotics and Automation*, volume 7, chapter Blanche: an experiment in guidance and navigation of an autonomous robot vehicle, pages 193–204. Springer Verlag, 1991.

[20] I. J. Cox and G. T. Wilfong. *Autonomous Robot Vehicles*. Springer Verlag, 1990.

[21] A. P. Dempster, A. N. Laird, and D. B. Rubin. *Journal of the Royal Statistical Society*, volume 39 of *B*, chapter Maximum likelihood from incomplete data via the EM algorithm, pages 1–38. 1977.

[22] F. DePaoli and F. Tisato. Architectural Abstractions and Time Modelling in Hyperreal. In *Proceedings of 7th Euromicro Workshop on Real-Time Systems*, June 1995.

[23] F. DePaoli and F. Tisato. On the duality between event-driven and time-driven models. In *Proceedings of 13th workshop on Distributed Computer Control Systems*, 1995.

[24] Jim des Rivières and Brian Cantwell Smith. The Implementation of Procedurally Reflective Languages. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 331–347, 2002.

[25] U. Dhond and J. Aggarwal. Structure from stereo - a review. In *Transactions on Systems, Man and Cybernetic*, volume 16, pages 1489–1510, 1989.

[26] U. Dhond and J. Aggarwal. Binocular versus trinocular stereo. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 2045–2050, 1990.

[27] G. Dissanayake, G. P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. In *IEEE Trans. Robotics and Automation*, volume 17(3), pages 229–241. IEEE, 2001.

[28] C. Dorai, G. Weng, A. K. Jain, and C. Mercer. *IEEE Trans. Pattern Anal. Mach. Intelligence*, volume 20, chapter Registration and integration of multiple object views for 3D model construction, pages 83–89. 1998.

[29] C. Dorai, J. Weng, and A. K. Jain. *IEEE Trans. Pattern Anal. Mach. Intelligence*, volume 9, chapter Optimal registration of object views using range data, pages 1131–1138. 1997.

[30] P. Drapikowski and T. Nowakowski. 3D Object Modelling in Mobile Robot Environment Using B-Spline Surfaces. 2002.

[31] C. Duchesne and J. Hervee. chapter A survey of Model-toImage Registration. Ellsevier Preprint, to appear.

[32] H. F. Durrant-Whyte, G. Dissanayake, and P. W. Gibbens. Toward deployment of large-scale simultaneous localisation and map building (SLAM) systems. In J. Hollerbach and Robotics Research The

Ninth International Symposium (ISRR99) D. Koditscheck, editors, editors, *IEEE Trans. Robotics and Automation*, pages 161–168. Springer-Verlag, 2000.

[33] H.F. DurrantWhyte. Uncertain Geometry in Robotics. In *IEEE Journal of Robotics and Automation*, volume 4(1), pages 23–31, 1988.

[34] C.R. Dyer. *Foundations of Image Understanding*, chapter Volumetric Scene Reconstruction from Multiple Views, pages 469–489. Kluwer, Boston, 2001.

[35] A.H. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the* $25^t h$ *IEEE International Conference on Software Engineering*, pages 149–159, May 2003.

[36] D. W. Eggert, A. Lorusso, and R. B. Fisher. *Machine Vision and Applications*, volume 9, chapter Estimating 3-D rigid body transformations: a comparison of four major algorithms, pages 272–290. Springer, 1997.

[37] A. Elfes. *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation.* PhD thesis, Department of Electrical and Computer Engineering - Carnegie Mellon University, 1989.

[38] A. Elfes. Sonar-based real-world mapping and navigation. In *IEE Journal of Robotics and Automation*, volume RA-3(3), pages 249–265, June 1989.

[39] S. Engelson and D. McDermott. Error correction in mobile robot map learning. In *Proceedings of ICRA-92*, 1992.

[40] O.D. Faugeras. *Artificial 3D Vision.* 1991.

[41] O.D. Faugeras. *Three-Dimensional Computer Vision (A geometric Viewpoint).* The MIT Press, 1994.

[42] H. Feder, J. Leonard, and H. Feder. Experimental analysis of adaptive concurrent localisation using sonar. In *Sixth Int. Symp. Experimental Robotics*, pages 213–222, 1999.

[43] M. Fowler and K. Scott. *UML Distilled.* Addison-Wesley, second edition, 2000.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[45] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why its hard to build systems out of existing parts. In *Proceedings of XVII ICSE.* IEEE, April 1996.

[46] D. Garlan and M. Shaw. *Advances in Software Engineering and Knowledge Engineering*, volume 1, chapter An Introduction to Software Architecture. World Scientific, 1993.

[47] G. Godin, M. Rioux, and R. Baribeau. Threedimensional registration using range and intensity information. In *Proceedings of SPIE Conf. Videometrics III*, volume 2350, pages 279–290, 1994.

[48] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley, 1993.

[49] C. Guerra and V. Pascucci. 3D segment matching using the Hausdorff distance. In *Proceedings of Image Processing and its Applications*. IEEE, 1999.

[50] J. S. Gutmann and C. Schlegel. Amos: Comparison of scan matching approaches for self-localization in indoor environments. In *Proceedings of the 1st Euromicro Workshop on Advanced Mobile Robots*. IEEE Computer Society Press, 1996.

[51] A. Guzzoni, A. Cheyer, L. Julia, and K. Konolige. Many robots make short work. In *AI Magazine*, volume 18(1), pages 55–64, 1997.

[52] J. Hertzberg and F. Kirchner. Landmark-based autonomous navigation in sewerage pipes. In *Proceeding of First Euromicro Workshop on Advanced Mobile Robots*. IEEE Computer Society Press, 1996.

[53] J.E. Hopcroft and J.D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, 1990.

[54] B. K. P. Horn. *J. Opt. Soc. Amer.*, volume 4, chapter Closed-form solution of absolute orientation using unit quaternions, pages 629–642. 1987.

[55] B.K.P. Horn, H.M Hilden, and S. Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. In *Journal of the Optical Society of America*, volume Series A, 5, 7, pages 1127–1135, 1985.

[56] P. Jensfelt and S. Kristensen. Active global localisation for a mobile robot using multiple hypothesis tracking. In *Proceedings of the IJCAI-99 Workshop on Reasoning with Uncertainty in Robot Navigation*, 1999.

[57] L. P. Kaelbling, A. R. Cassandra, and J. A. Kurien. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1996.

[58] P. Kahn, L. Kitchen, and E. Riseman. Real Time Feature Extraction: A Fast Line Finder for Vision-Guided Robot Navigation. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 12, pages 1098–1102, November 1990.

[59] R. E. Kalman. A new approach to linear filtering and prediction problems. In *Trans. of the ASME, Journal of basic engineering*, volume 82, pages 35–45. March 1960.

[60] B. Kamgar-Parsi and B. Kamgar-Parsi. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 19, chapter Matching Sets of 3D Line Segments with Application to Polygonal Arc Matching, pages 1090–1099. IEEE, October 1997.

[61] Gregor Kickzales, Jim des Rivières, and Daniel Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[62] B. Kuipers and Y.-T. Byun. A Robot Exploration and Mapping Strategy Based on a Semantic Hierarchy of Spatial Representations. In *Journal of Robotics and Autonomous Systems*, volume 8, pages 47–63, 1991.

[63] J. J. Leonard, G. Dissanayake, and H. F. Durrant-Whyte. Towards terrain-aided navigation for underwater robotics. In *Advanced Robotics*, volume 15(5), 2001.

[64] J. J. Leonard and H. F. Durrant-Whyte. *IEEE Transactions on Robotics and Automation*, volume 7, chapter Mobile Robot Localization by Tracking Geometric Beacons, pages 376–382. IEEE Transactions, 1991.

[65] J. J. Leonard and H. F. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *Proceedings of IEEE/RSJ International Workshop on Intelligent Robots and Systems IROS '91*, volume 3, pages 1442–1447, 1991.

[66] J. J. Leonard and H. F. Durrant-Whyte. Directed Sonar Sensing for Mobile Robot Navigation. In *Kluwer Academic Publisher*, 1992.

[67] J. J. Leonard, H. F. Durrant-Whyte, and I.J. Cox. Dynamic map building for an autonomous mobile robot. In *Journal of Robotics Research*, volume 11(4), pages 89–96, 1992.

[68] Y. Liu, R. Emery, D. Chakrabarti, W. Burgard, and S. Thrun. Using EM to lear 3D models with mobile robots. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2001.

[69] F. Lu and E. Milios. Robot pose estimation in unknown environments by matching 2d range scans. In *Proceedings of IEEE Computer Vision and Pattern Recognition Conference (CVPR)*, 1996.

[70] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings oF Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, October 1987.

[71] S. Manzoni, P. Mereghetti, D. Micucci, and V. Sachero. Knowledge-based support to indoor air quality control. In *Proceeding of Environmental IFIP Informatics Institute WG 5.11 5th International Symposium on Environmental Software Systems (ISESS).* IFIP conference series Environmental Software Systems Vol. 5., May 2003.

[72] F. Marchese. Numerical Potential Fields for Real-Time Robot Navigation. In *Proceedings of the 27th International Symposium on Industrial Robots (ISIR)*, pages 745–750, 1996.

[73] C. Martin and S. Thrun. Online acquisition of compact volumetric maps with mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[74] P. S. Maybeck. *Autonomous Robot Vehicles*, chapter The Kalman filter: An introduction to concepts. Springer Verlag, 1990.

[75] P. J. McKerrow. *Introduction to Robotics*. Addison-Wesley, Sidney, 1991.

[76] N. Medvidovic and R.N. Taylor. *IEEE Transactions on Software Engineering*, volume 26, chapter A Classification and Comparison Framework for Software Architecture Description Languages, pages 70–93. IEEE, January 2000.

[77] D. Micucci. Exploiting the Kaleidoscope architecture in an industrial environmental monitoring system with heterogeneous devices and a knowledge-based supervisor. In *Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering*, pages 685–688. ACM Press, July 2002.

[78] D. Micucci. An Object-Oriented software approach for a distributed human tracking motion system. In *Proceedings of Visual Communications and Image Processing (VCIP).* SPIE, July 2003.

[79] D. Micucci, M. Sarini, C. Simone, F. Tisato, and A. Trentini. Conceptual and concrete architectures in the design of cscw application. In *Proceedings of the annual Workshop on Agents.* Pitagora editrice, November 2002.

[80] D. Micucci, F. Tisato, and A. Savigni. Environmental monitoring systems: the Kaleidoscope architecture and the RAID case. In *Proceedings of WSDAAL 2001 Conference*, September 2001.

[81] D. Micucci and A. Trentini. A pattern-like framework to ease dynamically change components behaviour. In *Proceedings of the 15th international conference on Software Engineering and Knowledge Engineering.* ACM Press, July 2003.

[82] D. Micucci, A. Trentini, and F. Tisato. A connector-based approach for controlled data distribution in rtp architecture. In *Proceedings of 23rd International Conference on Distributed Computing Systems (ICDCS) - DDRTS 2003 Workshop.* IEEE Computer Society Press, May 2003.

[83] R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, January 1997.

[84] H. P. Moravec. *AI Magazine*, volume 9, chapter Sensor fusion in certainty grids for mobile robots, pages 61–74. 1998.

[85] V. Murino, A. Fusiello, N. Iuretigh, and E. Puppo. 3d mosaicing for environment reconstruction. In *Proceedings of 15th International Conference on Pattern Recognition*, pages 358–362, September, 3-7 2000.

[86] L. Nigro and F. Tisato. Timing as a programming-in-the-large issue. In *Proceedings of the JMCL '94*, 1994.

[87] I. Nourbakhsh, R. Powers, and S. Birchfield. *AI Magazine*, volume 16, chapter DERVISH an office-navigating robot. Springer Verlag, 1995.

[88] S. Oore, G. E. Hinton, and G. Dudek. *Neural Computation*, chapter A mobile robot that learns its place. 1997.

[89] F. De Paoli and F. Tisato. Architectural abstractions for real-time software. 1995.

[90] F. De Paoli, F. Tisato, and C. Bellettini. *Software Architectures: Advances and Applications*, chapter HyperReal: A Modular Control Architecture for HRT Systems. In Euromicro Journal of System Architecture, 1996.

[91] D.E. Perry and A.L. Wolf. *ACM SIGSOFT Software Engineering Notes*, volume 17, chapter Foundations for the Study of Software Architecture, pages 40–52. ACM SIGSOFT, October 1992.

[92] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[93] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[94] A. Savigni and F. Tisato. Kaleidoscope. A reference architecture for Monitoring and Control Systems. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA)*, 1999.

[95] A. Savigni and F. Tisato. Designing Traffic Control Systems. A Software Engineering Perspective. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems 8th Meeting of the Euro Working Group Transportation - EWGT)*, September 2000.

[96] A. Savigni and F. Tisato. Real-Time Programming-in-the-Large. In *Proceedings of ISORC 2000 The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 352–359, March 2000.

[97] B. Schiele and J. L. Crowley. A comparison of position estimation techniques using occupancy grids. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1994.

[98] A. Schultz, W. Adams, and B. Yamauchi. Integrating exploration, localization, navigation and planning through a common representation. In *Autonomous Robots*, volume 6. 1999.

[99] G. Shaffer, J. Gonzalez, and A. Stentz. Comparison of two range-based estimators for a mobile robot. In *Proceedings of the SPIE Conf. on Mobile Robots VII*, pages 661–667, 1992.

[100] H. Shatkay and L. Kaelbling. Learning topological maps with weak local odometric information. In *Proceedings of IJCAI*, 1997.

[101] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelisnik. *IEEE Transactions on Software Engineering*, volume 21, chapter Abstractions for Software Architecture and Tools that Support Them, pages 314–335. IEEE, April 1993.

[102] M. Shaw and D. Garlan. *Software Architecture. Perspective on an Emerging Discipline*. Pertice Hall, 1996.

[103] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1995.

[104] B.C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, 1984.

[105] Brian Cantwell Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[106] R. Smith and P. Cheeseman. On the representation and Estimation of Spatial Uncertainty. In *Int. J. Robotics Research*, volume 5(4), pages 56–68, 1986.

[107] R. Smith, M. Self, and P. Cheeseman. *In Autonomous Robot Vehicles*, chapter Estimating Uncertain Spatial Relationships in Robotics, pages 167–193. Springer-Verlag, 1990.

[108] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. Technical Report TR 4760 and 7239, 1985.

[109] Francisco Ortín Soler and Juan Manuel Cueva Lovelle. The nitrO Reflective Platform. In *Proceedings of International Conference on Software Engineering Research and Practice (SERP)*. CSREA Press, 2002.

[110] J. Stankovic. Misconceptions about real-time computing: a serious problem for next generation systems. *IEEE Computer*, October 1988.

[111] S. Thrun. Probabilistic algorithms and the interactive museum tour-guide robot Minerva. In *International Journal of Robotics Research*, volume 19(11), pages 972–999, 2000.

[112] S. Thrun. Probabilistic algorithms in robotics. In *AI Magazine*, volume 21(4), 2000.

[113] S. Thrun. A probabilistic online mapping algorithm for teams of mobile robots. In *International Journal of Robotics Research*, volume 20(5), pages 335–363, 2001.

[114] S. Thrun. Robot Mapping: a Survey. Technical Report CMU–CS–02–111, Carnegie Mellon University, Pittsburgh, PA, Pittsburgh, PA 15213, February 2002.

[115] S. Thrun, D. Fox, and W. Burgard. *Machine Learning*, volume 31, chapter A probabilistic approach to concurrent mapping and localization for mobile robots, pages 29–53. 1998.

[116] F. Tisato, A. Savigni, and W. Cazzola. Architectural Reflection. Realising Software Architectures via Reflective Activities. In *Proceedings*

of EDO2000 (2nd International Workshop on Engineering Distributed Objects), November 2000.

[117] W.M. Waite and A.M. Sloane. Software Synthesis via Domain-Specific Software Architectures. Technical Report CU-CS-611-92, University of Colorado at Boulder, September 1992.

[118] M. W. Walker, L. Shao, and R. A. Voltz. *CVGIP: Image Understanding*, volume 54, chapter Estimating 3-D location parameters using dual number quaternions, pages 358–367. 1991.

[119] X. Wang, Y. Q. Cheng, R. T. Collins, and A. R. Hanson. Determining correspondences and rigid motion of 3-d point sets with missing data. In *Proceedings of IEEE Conf. Computer Vision Pattern Recogn*, pages 252–257, 1996.

[120] G. Wei, C. Wetzler, and E. von Puttkamer. Keeping track of position and orientation of moving indoor systems by correlation of range-finder scans. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1994.

[121] R. T. Whitaker. *International Journal of Computer Vision*, volume 29, chapter A level-set approach to 3D reconstruction from range data, pages 203–231. October 1998.

[122] S. B. Williams, G. Dissanayake, and H. F. Durrant-Whyte. Towards multi-veicle simultaneous localisation and mapping. In *Proceedings of International Conference on Robotics and Automation*, pages 2743–2748. IEEE, 2002.

[123] B. Yamauchi. Mobile robot localization in dynamic environments using dead reckoning and evidence grids. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1996.

[124] B. Yamauchi and P. Langley. Place recognition in dynamic environments. In *Journal of Robotic Systems*, volume 14(2), pages 107–120, 1997.

[125] Z. Zhang. *International Journal of Computer Vision*, volume 13, chapter Iterative point matching for registration of free-form curves and surfaces, pages 119–152. 1994.