Instituto de Sistemas e Robótica

Instituto Superior Técnico

AGENT-BASED SOFTWARE ARCHITECTURE

TECHNICAL REPORT

RESCUE – COOPERATIVE NAVIGATION FOR RESCUE ROBOTS

FCT SRI/32546/99-00

JOÃO PEDRO FRAZÃO

December 2003

Contents

1 Introduction

This report describes an Agent Based Software Architecture that intends to close the gap between hybrid systems and software agent architectures [7] [8] [9]. The described tool provides support for task design, task planning, task execution, task coordination and task analysis for a multi- robot system.

There are currently available several software tools for the mission design and development for teams of real robots like TeamBots [13], Mission Lab[12] and CHARON[11]).

TeamBots is a collection of Java application programs and libraries designed to support multiagent systems. It supports simulation of robot control systems and execution of the same control systems on mobile robots. It includes a communication package (RoboComm), and Clay, a library to support behavior-based control systems. The simulation environment is written entirely in Java. Execution on mobile robots sometimes requires low-level libraries in C, but Java is used for all higher-level functions.

Mission Lab is a mission specification software that uses visual programming and reusable components. It is composed by several subsystems such as console display, a visual configuration editor, a simulator, and a runtime and usability data logging module. MissionLab generates code that runs under a distributed architecture (e.g., the main user's console can run on one computer while multiple robot control executables are distributed across a network, potentially on-board the actual robots they control.).

CHARON is a language for modular specification of interacting hybrid systems based on the notions of agent and mode. It provides operations for both an hierarchical description of the system architecture (referring to the agents relations), and an hierarchical description of the behavior of an agent. The discrete and continuous behaviors of an agent are described using modes. A mode is basically a hierarchical state machine, that is, a mode can have submodes and transitions connecting them. Agents in CHARON can

communicate via shared variables and communication channels. Both event-driven discrete state and time-driven continuous state system descriptions are supported.

Another framework for generic and reusable robotic components that can be adapted to a number of heterogeneous robot platforms has been developed in recent years at the Jet Propulsion Laboratory, under the designation "Coupled Layered Architecture for Robotic Autonomy" (CLARAty). CLARAty consists of two distinct layers: a Functional Layer and a Decision Layer [10].

The lower abstraction Functional Layer encapsulates functions which interface with the actual hardware or simulations and provides low to mid-level autonomy. The higher abstraction Decision Layer provides high-level autonomy and is concerned with mission constraints and resources. The interface between the two is based on an intentional overlap between the two layers, allowing the higher declarative level to use high resolution information and the lower functional level to build abstractions from its basic functionalities, such as a navigator. There is a clear focus on software modularity and reusability, as well as on abstracting the functionalities of distinct robotic platforms. The architecture does not seem to be designed for cooperative multi-robot systems.

The report is divided in three major parts:

First, we present, in Section 2, the conceptual model of the proposed Agent Software Architecture that supports the work currently being developed in the “Rescue - Cooperative Navigation for Rescue Robots” project [15].

In Section 3, a reference guide to the application program interface (API) and to the software architecture is presented. This reference guide is targeted for researchers and students working on the Rescue Project, as well as to future users of the architecture.

Finally, in Section 4, we describe the Rescue project instantiation of the concepts involved in this software architecture.

2 Conceptual Model

The conceptual model of the agent-based software architecture includes different types of agents that can be combined both hierarchically and in a distributed manner [1].

The architecture support information fusion between several sensors and the sharing of information between the agents by a **Blackboard** [2**]** and is geared towards the cooperation between robots.

Agents are generically organized hierarchically. At the top of the hierarchy, the algorithms associated with the agents are likely to be planners, whilst at the bottom they are interfaces to control and sensing hardware. The planner agents are able to control the execution of the lower level agents to service high-level goals. The latter can be distributed across several processors and/or robots. To offer platform independence, only the lowest level agents are specific to the hardware, and these have a consistent interface for communication with the planning agents that control their execution.

The elements of the architecture are the **Agents**, the **Blackboard**, and the **Control/Communication Ports**. Next, each of them is described in detail.

2.1 Elements

2.1.1 Agent

We define **Agent** as an entity with its own execution context, its own state and memory and mechanisms to sense and take actions over the environment.

Rescue Agents have a control interface used to control their execution. The control interface can be accessed remotely by other agents or by a human operator. Through the control interface, an Agent can be enabled, disabled and calibrated.

Agents share data by a data interface. Through this interface, the agents can sense and act over the world.

There are two main classes of agents, *Composite Agents* and *Simple Agents*.

- The Composite agents are Agents that are composed by two or more agents. The principle behind composite agents is to abstract a group of related agents. An agent society can have several types of groups. Groups represent the way that agents relate or interact with each other's. Composite agents allow a group of agents to be faced as a single agent by designers, by operators or by other parts of the system. For this to be possible, a composite agent must take control over the agents that compose him. Moreover, composite agents must be easy to use: their usage should be only a matter of choosing the right type of composite agent and then plugging the controlled agents.
- Simple agents are agents that do not control other agents; they do not even need to know about the existence of other agents. Simple agents represent hardware devices, data fusion and control loops.

For now the supported agent types are:

  - `Goal-Based Agent:` a composite agent that knows other agent's actions, the context in which to apply them and the expected result of the actions to build plan(s) to reach the proposed goal, this type of agent is foresee but not currently implemented.
  - `Finite State Machine Agent:` a composite agent used to model complex interactions between agents. Each plugged Agent is associated to a state of the FSA. The execution of the FSA given an input sequence of events makes the FSA go through a sequence of states. In each of the states, the associated controlled Agent is executed. Therefore, the state sequence implies the sequential execution of the controlled agents.
  - `Concurrent Agent:` composite agent that represents the simultaneous execution of two or more agents. All the agents plugged on this composite agent will execute simultaneously.
  - `Exclusive Agent:` This composite agent represents the exclusive execution of agents. It is used to make sure that only one of the plugged

agents is executing at a given time. This is a type of agent similar to the micro-agents of the SocRob project [3].

- `Periodic Agent:` An agent that executes a given function periodically. The period is specified. This agent can be used for data fusion and control loops.
- `Sensor Agent:` A driver or a server to a hardware device of the sensor type. These are customized made for each type of sensor. Usually they take data from the sensor to the blackboard.
- `Actuator Agent:` A driver or a server to an hardware device of the actuator type. These are customized for each type of actuator. Usually they take commands from the blackboard to the actuator.

The possible combinations among these agent types provide the flexibility required to build a Mission for a cooperative robotics project, such as the Rescue project [1]. For special interactions that are not currently supported, the architecture is open to include other types of agents.

We refer to the mission as the top-level task that the system should execute. In the same robotic system, we can have different missions. The robotic system runs a mission similarly to an operating system running an application. The mission is a particular agent instantiation. The agent's implementation is made in a way to promote the reusability of the same agent in different missions [1].

2.1.2 Blackboard

The **Blackboard** is a distributed structure that gives support to the data exchange between the Agents. Each entry on the blackboard is a collection of samples ordered by their creation time. Since all the data shared between the agents goes through the blackboard, reads and writes are concurrent to maximize performance.

2.1.3 Ports

Ports are a handy abstraction to keep the agents decoupled from other agents. When an agent is defined, his ports are kept unconnected. This approach enables using the same agent definition in different places and in different ways. There are two types of ports: *control ports* and *data ports* (Figure 1).

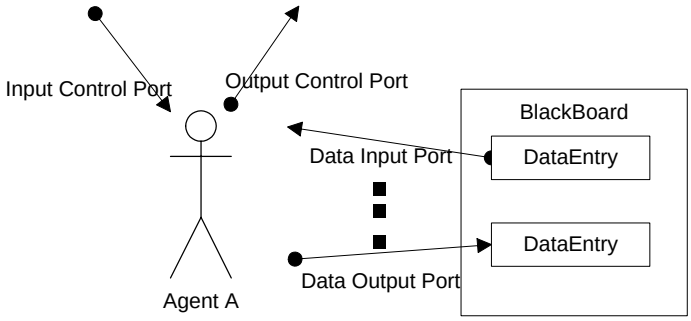

Figure 1 – Agent Control and Data Ports.

Control ports are used within the Agent hierarchy to control agent execution.

Each agent is endowed with one upper control interface. The upper interface has two defined control ports. One of the ports is the *input control port*; we can see it like the request port from where the agent receives notifications of actions to perform from higher-level agents. The other port is the *output control port* through which the agent reports progress to the high level agent. This is what we denote as a consistent interface for control.

Composite agents also have a lower level control interface from where they can control and sense the agents beneath him. The lower level control interface is customized in accordance to the type of agent. For instance, a `Finite State Machine Agent` has as many lower level control ports as agents that he is controlling. An additional data input port is used to enable the agent receiving events (Figure 2).

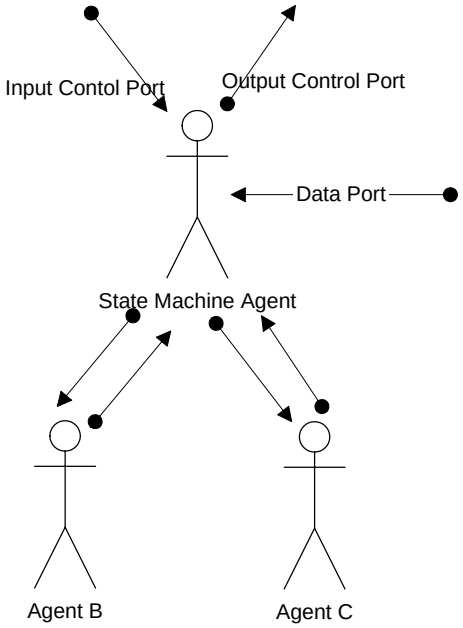

Figure 2 – A Composite Agent and two controlled agents beneath him.

Data ports are used to connect the agents to the *blackboard data entries*, enabling agents to share data. More than one port can be connected to the same data entry. Several agents can be reading from the same place at the same time (Figure 3). However if a data entry has more than a write port connected, some sort of contention resolution mechanism (such as a `Exclusive Agent`) must be used.

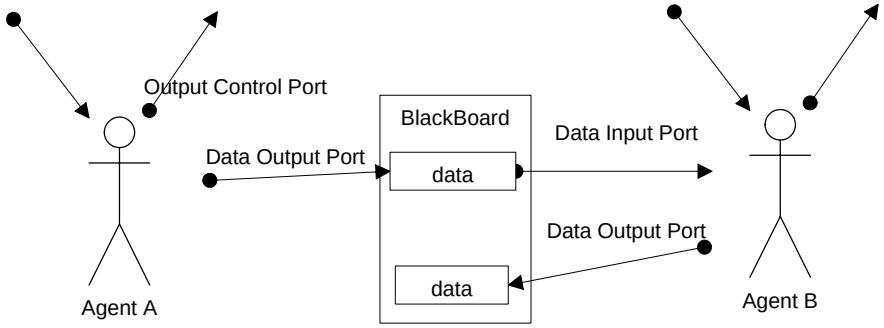

Figure 3 – Agent A is writing a value on the Blackboard that Agent B is reading.

The data ports are linked together through the blackboard. For configuration flexibility of the agent's hierarchy, the agent ports are not assigned in the definition of the agent. Ports are assigned in the instantiation of the agent hierarchy.

Each agent also defines a new scope (his scope) inside of the blackboard. The *scope* can be viewed as the context of the agent.

2.2 Execution Modes

Traditionally, in Robotics, there is a trend towards giving importance only to the run-time impact of the robotic system architecture. Unfortunately, during several research phases, robots are stopped most of the time. Much time and resources are consumed in system design, system calibration and system analysis. These are very relevant issues often forgotten by Robotics researchers. A well-designed architecture targets the support and speed-up of these development phases [16].

Usually, properties such as system distribution and concurrency are relevant during the mission execution, since they provide better resource allocation and robustness.

Centralization and persistency are important properties when dealing with the robots prior to the mission execution or handling the data acquired after the mission execution. Those properties also help managing different missions for a team of robots.

Even during mission execution, system distribution is not required all the time for all the aspects. To control the robots it is better to think of them as a fleet, and to be able to exert control over the fleet from a central place, when needed.

Under this architecture, a different **execution mode** exists for each development phase of a multi-robot system**.**

The system hardware is composed by a central station and by the robots. Despite the fact that some robots do not have a computer onboard (e.g., the blimp in the Rescue project), it is understood that there is a computer that controls such robot(s). The robots and the central station use a wireless network for communication.

The centralized **execution modes** of the software architecture are located on the central station. In spite of being centralized, they do interact with the robots (Figure 4).

The control mode follows a distributed approach. This mode is spread across the robots, however can be controlled from the central station (Figure 4).

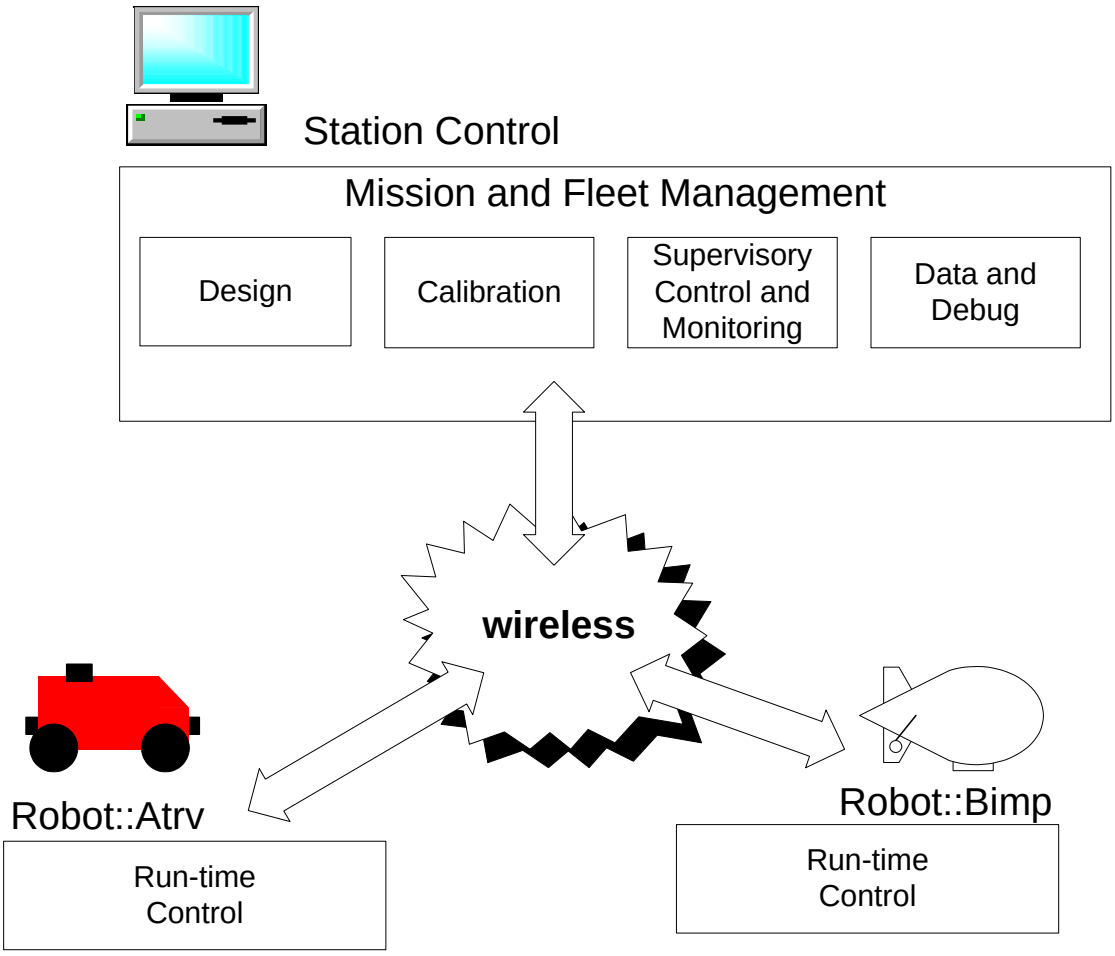

Figure 4 – System Execution Modes – Example for the rescue project.

Next, we describe each of the five **execution modes** available on the Software Architecture.

First, we describe the **Control Mode** that refers mostly to the run-time interactions between the **elements**. Afterwards, we describe the **Design Mode**, the **Calibration Mode**, the **Supervisory Control Mode** and finally the **Logging and Data Mode**.

2.2.1 Control Mode

The control exerted by an upper-level agent over a lower-level agent is accomplished through special and well-defined functions: `start, stop, set` and `reset`. In this sense if we `stop` the agent that encapsulates the whole fleet, he will request his lower-level agents to stop, so a cascading reaction will stop all the agents' hierarchy inside each of the robots, from the top down to the lowest level hardware agents, including the robots. A similar behavior happens with the `start` command.

2.2.2 Design Mode

The Design Mode is similar to a graphics drawing program. In these programs, there are different tools for the different graphic objects, such as lines, squares and so on.

In the Design Mode, instead of drawing tools for each type of graphic we have a drawing toolbox for each type of the supported agents, plus one additional toolbox for linking agents written in pure code. The output is a meta-language that represents an instantiation of the supported agents or the included code files when the agent is implemented in pure code. The language describes the connections between the agents. This meta-language is then transferred to the target robots for execution.

Currently we are developing the language. All the support to the language instantiation has been developed.

2.2.3 Calibration Mode

Usually, robots have controllers, sensory processing and even hardware that must be configured or calibrated. Controllers, behaviors and perceptual processes have parameters that must be tuned. Usually this data is kept in text files for ease of modification without the need to recompile the code. For more complex calibration procedures (like color segmentation) special configuration process must be executed sometimes.

To simplify the calibration procedure for the robot fleet, each agent has an associated calibration window, which can be requested remotely before the start of the mission. The

calibration data is persistent and can be used in a later mission. To keep management of the fleet a simple job, the calibration data is stored in the central station. This data is distributed to the robots before run time.

The operator does the calibration following the instructions appearing in the remote window. For each agent involved the mission, the program asks the operator if he/she wishes to make a new calibration, to skip, to save or to load a previous one. This is done in a top-down manner. Answering “skip” to the agent that encapsulates the whole fleet will produce the result of all robots with all their agents being calibrated by the latest data used.

This mode provides support on managing the data calibration files. It also supports the way the various data types are written and read from the files.

2.2.4 Supervisory Control Mode

Each of the agents has, in addition to the Calibration window, an associated Supervisory Control window, corresponding to the Supervisory Control Mode, designed to be user-friendly. Therefore, the agent that controls the motors has an user-interface appropriated for its specific task. This interface is different from the user-interface to a (higher-level) planner agent. There are common features to all agents like the request to start, the request to stop or the request to logging. All the common features are provided by the mode window in the form of buttons and text boxes.

The supervisory control window uses the same program-interface through which the agents receive control requests from higher-level agents and get data from the same program-interface through which the agents report success, failure or progress to the higher-level agents. The only difference is the use of a graphical window for ease of human use. If the operator chooses to control an agent from the hierarchy, the framework should disable all control requests arriving at the controlled agent from other agents.

In the supervisory control window, there is also a *blackboard view*. In the blackboard view, the supervisor can consult or modify the various types of variables. This is an extension of the blackboard view interface of the SocRob project [14].

2.2.5 Logging and Data Mode

Each of the agents can keep a logging file. if the supervisor chooses an agent whose activities are to be logged, that file is written locally inside each of the robots. After the mission ends, the log files are stored in the central station. During run-time, an operator can also choose to consult the logging of a particular agent. This mode logs, with the corresponding time tag, all the requests arriving and all reports departing an agent. Changes in the variables inside the blackboard can also be selected to be automatically logged by the framework with the corresponding time tag. Additional logging should be made inside the code of the agent. The framework provides a program interface for doing so, without the need of opening files and managing files.

3 Application Programming Interface Reference

The concepts presented in this software architecture follow an Object Oriented approach. This project was developed using C++ and CORBA. We have chose C++ because it gives the best compromise between performance and the Object Oriented flexibility. Whenever more flexibility was needed, CORBA was used. CORBA was also used for the communication between the robots. We have chosen to give preference to the C++ standard library. For the future developers, namely these working on image processing, it is advised the use of the OpenCV library from Intel. To connect the system to one of the robots we have used the iRobot mobility library [4] [5] [6].

3.1 Blackboard

The blackboard is a shared memory system. The blackboard can store several types of data variables:

```
FloatVar
FloatVectorVar
IntVar
IntVectorVar
```

`CharVar`

`CharVectorVar`

Usage:

```
FloatVectorVar  Velocity("RobotVelocityXYTheta",3,10);
```

This line creates a data entry called "`RobotVelocityXYTheta`" that is a vector composed of three Floats. The data entry can hold a history of ten velocities.

```
BBAddEntry(&DataEntry);
```

This method adds a previously created data entry to the blackboard.

3.2 Agents

All the agent types share a common set of features, so we have an abstract class `Agent` that represents and implements those features. Some of the methods described here are CORBA methods; this means that they can be called over the network [4].

First the Agent is created with the `New(`*`name`*`)` method, and the memory is allocated. Next, the framework calls the `Initialize()` function that initializes the execution context of the agent.

In the Stopped state, the agent is prepared for execution but remains stopped. Here, he can receive a Calibration event and start the calibration process. The Agent can also receive the `StartActivity` event. When that happens, the `OnEntry()` function is called once and after that the Agent will run the `Activity()` function until he receives a `StopActivity` event. Where the framework calls the `OnExit()` function once and then the agent returns to the `Stopped` state.

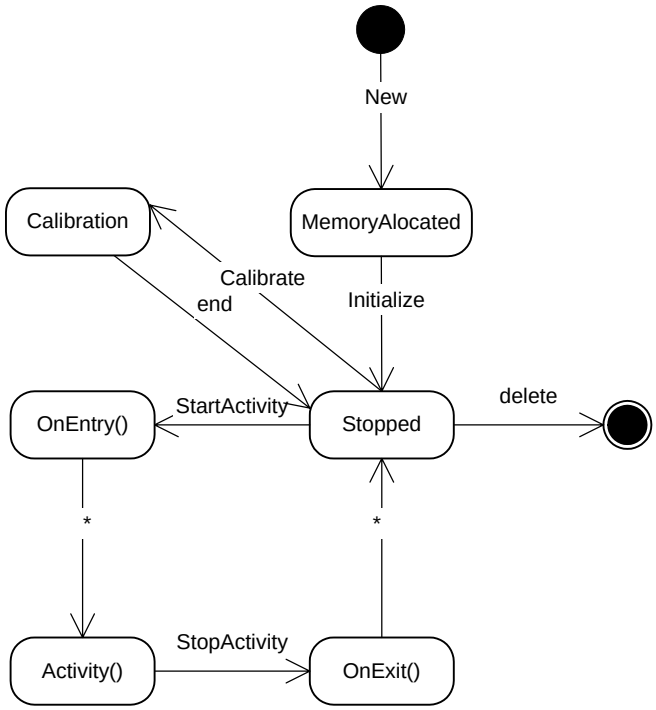

Figure 5 - The life cycle of an Agent

The agent can only be destroyed on the `Stopped` state.

Each agent has a uniquely identified *`name`* given when the agent is created.

Methods:

```
Virtual Initialize()

Virtual OnEntry()

Virtual Activity()

Virtual OnExit()

Virtual Calibrate()
```

3.2.1 Control Ports

The purpose of the control ports is to control the execution of the agent. All of the available functions can be called through the network. Control port operations are represented by functions, given that all the agents have always the same type and number of control ports.

`StartActivity (`*`timeout`*`)`

This function starts the agent activity. This is a handshaking protocol. The function is blocking until the *timeout* elapses. If after the *timeout* the called agent does not start his activity then the caller Agent is released. The *timeout* value should be set to the maximum expected communication delay.

This function throws a `CommunicationError` exception in the case of a timeout.

This function is a hierarchical function (see Composite Agents).

`StopActivity (`*`timeout`*`)`

This function stops the agent activity. This is a handshaking protocol. The function execution is blocked until the *timeout* elapses. If, after the *timeout*, the called agent does not stop his activity, then the caller Agent is released. The *timeout* value should be set to the maximum expected communication delay.

This function throws a `CommunicationError` exception in the case of a timeout.

This function is a hierarchical function (see Composite Agents).

`Calibrate (`*`timeout`*`, `*`file`*`)`

Calling this function starts the calibration process of the Agent. If, after the *timeout*, the agent did not start the calibration process, the function throws a `CommunicationError` exception, and the caller will be released. *File* is the file

from where the agent reads the calibration data. If *file* is null, the agent will start the interactive calibration process.

This function is a hierarchical function (see composite agents).

```
StartLog();

StopLog();

SetLogFile(Path/File);
```

3.2.2 Data Ports

An Agent receives and sends blackboard data through the data ports. The ports are required to make an indirection which decouples the agent implementation from the agents use.

Every agent has a *port table* that connects a given `PortName` to a given `BBentry`. There are several types of data ports, such as `integer`, `float` and `char`. All these types exist in vector form too.

Port Types:

```
FloatPort

FloatVectorPort

IntPort

IntVectorPort

CharPort

CharVectorPort
```

Declaration example:

```
FloatVectorPort *VelocityPortPointer;
```

Reading:

```
GetSampleByTime(timeStamp,&sample)
GetSampleBySeq(int,&sample)
GetSampleByOrder(int,&sample)
GetSampleLast(seqnumber,&sample)
```

Writing:

```
PutSample(&sample)
```

AgentPortsMethods:

```
AddNewPortToAgent("PortName",Portpointer)
BindPortToBBEntry("AgentPath/PortName",
DataEntryPointer);
```

3.3 Agent Types

3.3.1 Concurrent Agent

This is a Composite Agent. The purpose of this agent is to run agents in parallel: it should be used when two or more agents share the same life period.

When a hierarchical method is called on the concurrent agent, the agent calls the corresponding methods of all the agents added to him.

`AddAgent(`*`Agent`*`)`

This method adds an *Agent* (any type of agent) to the Concurrent Agent. This method automatically connects the control ports of the agents. This method can only be called between the `new()` constructor and the `Initialize()` method.

3.3.2 Exclusive Agent

This is a Composite Agent. The purpose of this agent is to run the added agents one at a time: It should be used when two or more agents share the same life period but they should not be executed simultaneously. This agent can be used to model some sort of exclusion mechanism.

This agent is similar to the SocRob project micro-agents that run one *plugin* of a set of *plugins* at a time. The Exclusive Agent chooses which agent to run when he receives the event associated with that Agent.

The Exclusive Agent has a data port called `Select` to receive the events.

`AddEventAgent(`*`event, agent`*`)`

This method adds an *agent* (of any type) to the Exclusive Agent. The added *agent* is associated with the *event* given. Whenever this *event* is detected the *agent* is started.

3.3.3 Finite State Machine Agent

This is a Composite Agent. The purpose of this agent is to execute a different agent in each state of a state machine. Whenever a state is reached, the Agent associated with that state is started.

The Finite State Machine Agent has a data port called `Event` to receive the events.

The Finite State Machine Agent has a special port called `State`. Whenever a Finite State Machine Agent changes the port `State`, it is updated with the new state.

`AddState (agent)`

This method adds an *agent* (of any type) to the Finite State Machine Agent. This Agent is associated with a state, and the name of the state is the name of the agent.

`AddTransition (event, state1, state2)`

This method adds a transition (triggered by *event*) from *state1* to *state2*

`MarkFirstState (state)`

This method marks *state* as the starting state of the Finite State Machine.

3.3.4 Periodic Agent

This agent runs the virtual method `PeriodicActivity()` in a loop with the period chosen. The user should override this method with his/her own function.

He can also override the `OnEntry()`, `OnExit()` and `Initialize()` functions.

`SetPeriod(timeval);`

This function sets the period *timeval* for the `PeriodicActivity`.

`Virtual PeriodicActivity();`

The framework calls this function periodically. The user should redefine this function with the required functionality. This function is already called in loop. This is similar to the `method function` of the SocRob Project.

3.4 Using the API

3.4.1 Defining a Periodic Agent

In this section, we give a simple example of how to define a periodic agent:

`#include “Agents.h”`

include for the Abstract class Periodic Agent

```
class MyAgent:public PeriodicAgent

FloatPort *VelocityPortPointer; //port declaration

Float velocitySample; // sample declaration


MyAgent() {
```

On the constructor we give a name to the ports:

```
AddNewPortToAgent("PortName", VelocityPortPointer);

}
virtual PeriodicActivity {
```

This function is called in loop.

First, in the beginning of the loop step, we should read the BB variables.

Reading from the port:

```
VelocityPortPointer->GetSampleLast(velocitySample);
```

User code: here we should put the code

/* CODE HERE */

At the end of the loop step, we should actualize the BB variables:

Write a value to the data-port:

```
VelocityPortPointer->PutSample(3);

}
```

3.4.2 Building an Agent Hierarchy.

In this section we give a simple example of how to instatiate agents and build a small hierarchy (Figure 6).

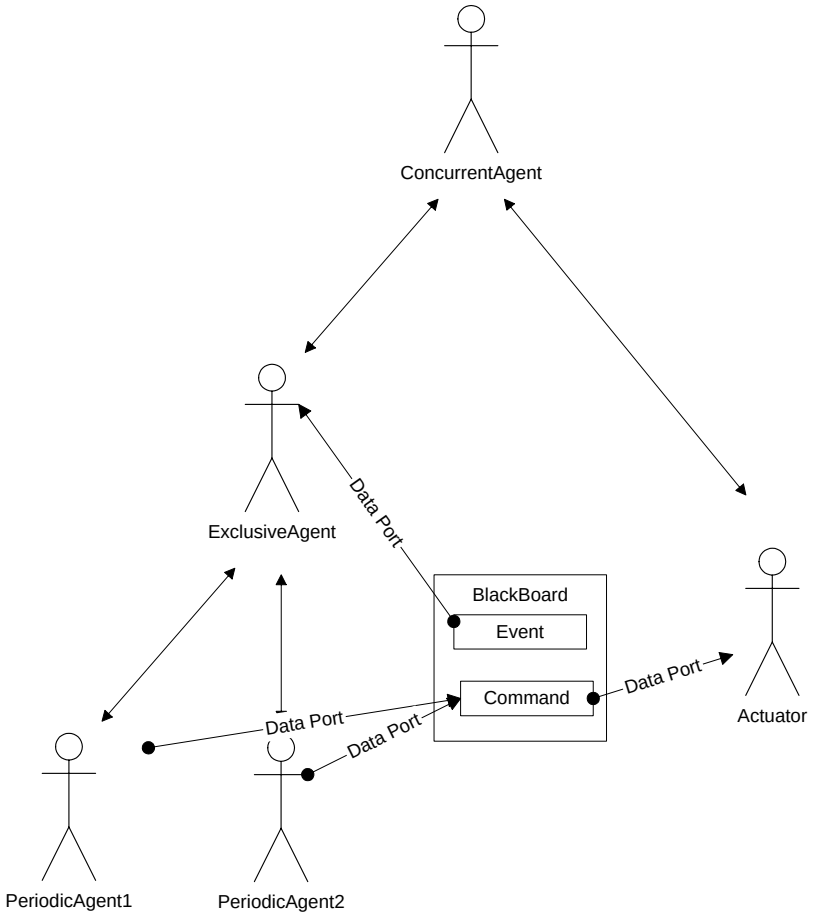

Figure 6 – Small agent hierarchy

In this example the `ExclusiveAgent` and the `Actuator` agent run in parallel because they are "inside" a concurrent agent. The actuator agent picks a `command` from the Blackboard that it sends to the hardware atuator device.

Inside the Exclusive agent there are two periodic agents that send in the example, velocity `commands` to the `actuator` agent. The active periodic agent is chosed by writing an event on the blackboard entry called "`event`". The `ExclusiveAgent`, upon detection of `event1` starts the agent `PeriodicAgent1`, when he detects the `event2` he stops the `PeriodicAgent1` and starts the `PeriodicAgent2` and so on.

Intanciating the periodic Agents:

```
MyAgent PeriodicAgent1(“PeriodicAgent1”);

MyAgent PeriodicAgent2(“PeriodicAgent2”);
```

Intanciating a Exclusive Agent:

```
ExclusiveAgent varExclusiveAgent(“ExclusiveAgent”);
```

Inserting two Periodic Agents on the Exclusive Agent

```
varExclusiveAgent.AddEventAgent(“Event1”,PeriodicAgent1);

varExclusiveAgent.AddEventAgent(“Event2”,PeriodicAgent2);
```

Intanciating an Concurrent Agent

```
ConcorrentAgent varConcorrentAgent(“ConcurrentAgent”);
```

Inserting one Actuator Agent and a Exclusive Agent to the Concurrent Agent:

```
varConcurrentAgent.AddAgent(ExclusiveAgent);

varConcurrentAgent.AddAgent(ActuatorAgent);
```

Intanciating the data entries on the Concurrent Agent blackboard:

```
FloatVar VelocityEntry(“velocity”,1);

ConcurrentAgent.BBAddEntry(&VelocityEntry);

charVectorVar Event(“event”,256,1);

ConcurrentAgent.BBAddEntry(&Event);
```

Connecting the Periodic Agents Ports together

```
ConcurrentAgent.BindPortToBBentry(“AnExclusiveAgent/Periodi
cAgent1/VelocityPort”,VelocityEntry);

ConcurrentAgent.BindPortToBBentry(“AnExclusiveAgent/Periodi
cAgent2/VelocityPort”,VelocityEntry);
```

```
ConcurrentAgent.BindPortToBBentry(“AnExclusiveAgent/select”
,Event);
```

If the agent hierarchy is not started the agents do not do any kind of work.

To start the agent hierarchy we simple need to start the Concurrent Agent:

```
ConcurrentAgent->start();
```

To change to periodic Agent1:

```
ConcurrentAgent->EventPort->PutSample(“Event1”);
```

To change to periodic Agent2:

```
ConcurrentAgent->EventPort->PutSample(“Event2”);
```

Finally, to stop the agent hierarchy we simple need to stop the Concurrent Agent:

```
ConcurrentAgent->stop();
```

4 Agent Architecture Applied to the Rescue Project

Under the reference scenario of the Rescue project, the land robot should be able to build a topological map, and be able to locate itself on that map, as well as to show different navigation capabilities, such as topological navigation with obstacle avoidance. With the topological navigation the robot should be able to go from one topological state to an arbitrary topological state. He should be able to change from Topological Navigation to either Metric Navigation or User Operated Navigation. All the values on the blackboard of the land robot can be read over the network.

The following sections describe the top-down instantiation of such a Rescue Mission.

4.1 Top Level Agents

The Figure 7 shows the first system decomposition. Sensor and Actuator agents where kept out of the diagram for the ease of interpretation. In addition to one agent per sensor and one per actuator, the system is split into five main Agents.

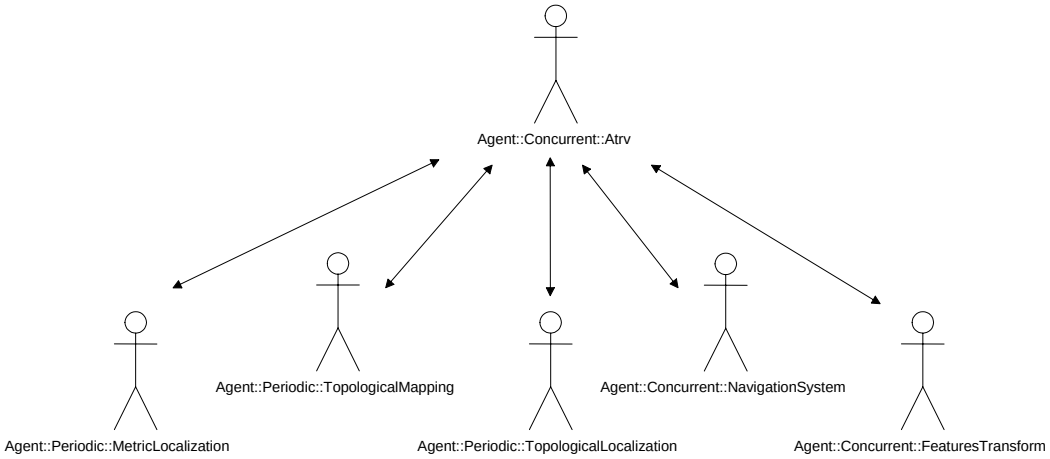

Figure 7 – System Top Agent

Each of the agents is responsible for a subsystem:

- `Features Transform` – This group of agents is responsible for picking raw data from the several sensors (sonar, laser, compass and image). The raw data is subsequently transformed into features that the Topological agents can use [17][18].
- `Navigation System` – This group of agents is responsible for the either the topological or the metric navigation of the robot. The Navigation sub-system includes obstacle avoidance behavior.
- `Topological Localization` – This agent gets the data-features and, comparing then with information taken from the topological map, determines where the robot is on the topological map [17][18].
- `Topological Mapping` – This agent is responsible for picking the features and building the topological map [17][18].
- `Metric Localization` – This agent is responsible for picking raw data from several sensors (odometric, GPS and compass). This data is fused together to determine the robot metric position and velocity.

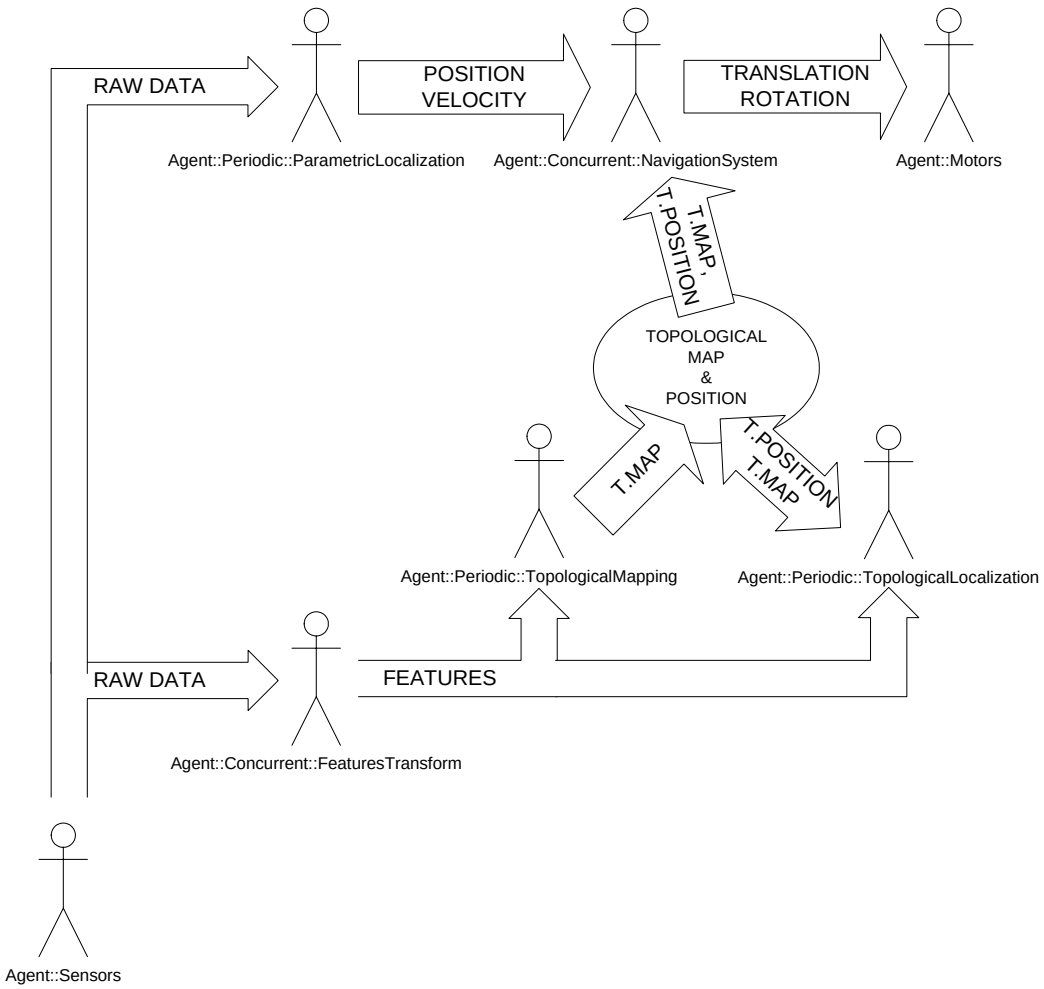

Figure 8 – Data Flow for the first system decomposition.

The Figure 8 depicts the data exchanged by the top rescue agents.

The arrows represent data connections between the agents. As explained before these connections are made throughout the blackboard. Arrows represent more than a value being exchanged. When an arrow is simple, it means that the starting agent is writing the data on blackboard entries. The pointed agent is reading the data from the blackboard entries. Only the `TopologicalMap` and `TopologicalPosition` data entries are represented.

If the arrow forks it means that more than one agent is reading the same data. When the arrow is double it means that the agent is reading and writing data. In the diagram, the `TopologicalLocalizationAgent` is reading the `TopologicalMap` data and writing the `TopologicalPosition` data.

4.2 Bottom Agents.

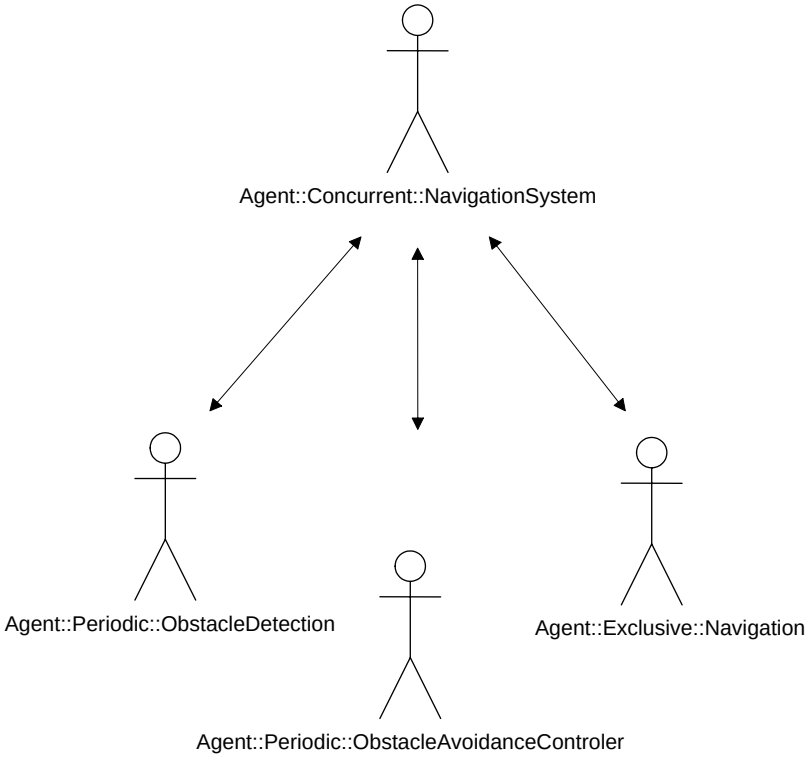

Figure 9 – Navigation System decomposition.

The navigation system includes an `ObstacleAvoidanceController` agent, an `ObstacleDetection` agent, and a `Navigation` agent. The `Navigation` agent is an `Exclusive` agent that multiplexes the several Navigation behaviors.

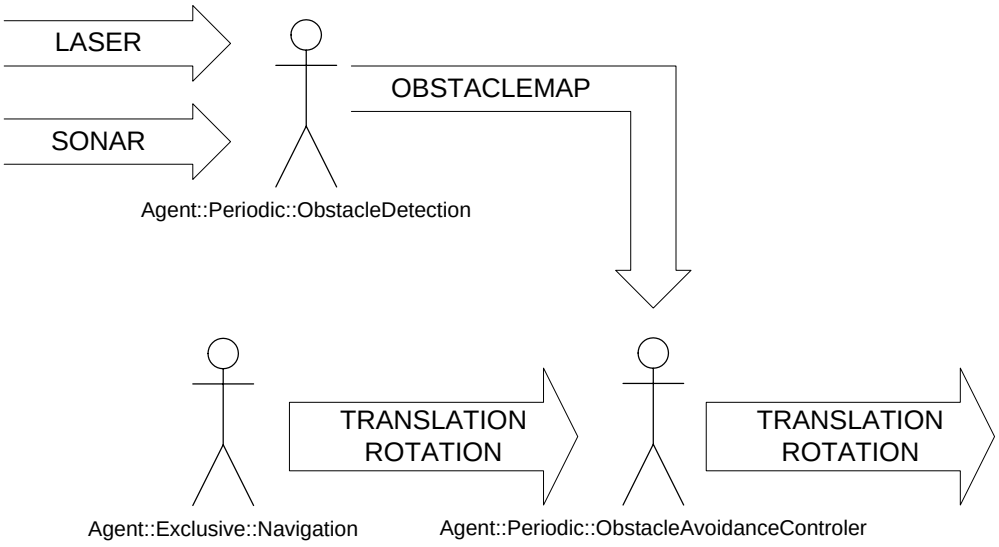

Figure 10 – Data exchange inside the Navigation System.

The `Exclusive Agent Navigation` chooses the desired robot `translation` and `rotation`. The `ObstacleAvoidance` controller chooses the possible Translation and Rotation given the `ObstacleMap`. The `Laser` data and `Sonar` data come from the `Sensor` agents. The translation and rotation commands end on the `Motors` agent (see Figure 8).

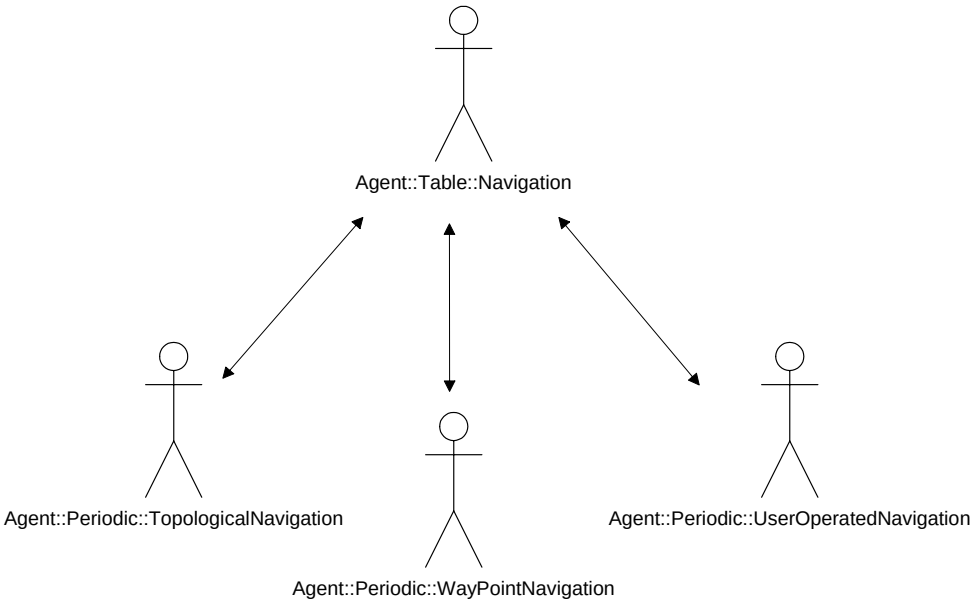

Figure 11 - Navigation Behaviors inside the Exclusive Navigation Agent

Since the agents are inside a Exclusive Agent this means that they never execute simultaneously.

- `Topological Navigation` –This agent drives the robot towards the desired objective state with the information collected from the `TopologicalMap` [17][18].
- `WayPointNavigation` – This agent receives a `WayPoint` list and controls the robot to follow the path.
- `UserOperatedNavigation` – This agent receives commands given by the user to drive the robot.

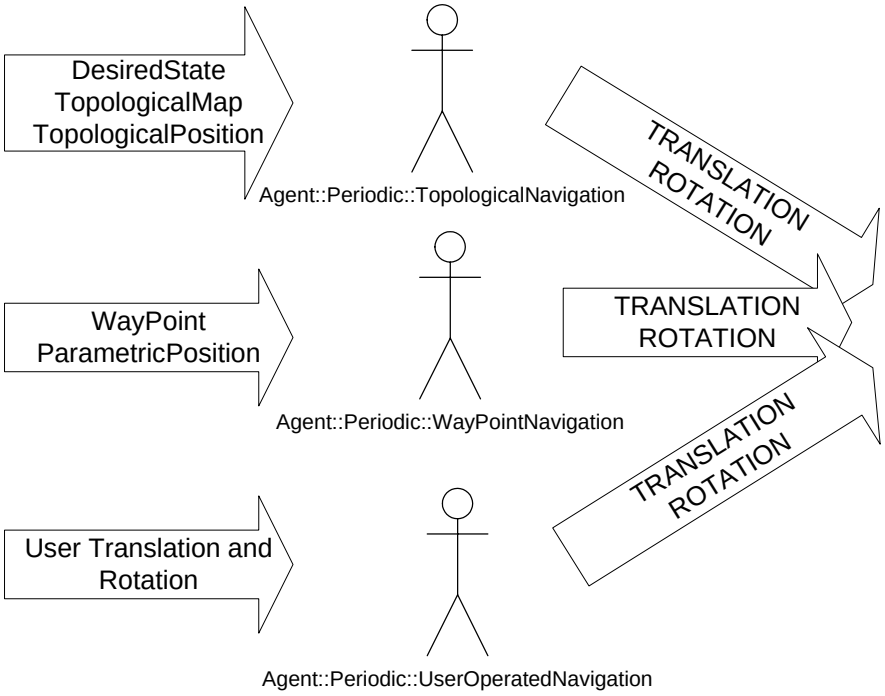

Figure 12 - Data flow inside the Navigation Agent.

From Figure 12 its should be clear why the Navigation agents have to be inside the Exclusive agent; All the agents are writing to the same blackboard entry.

5 References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. “Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley, Reading, MA, 1995.

[2] Hays-Roth, B., “A Blackboard Architecture for Control”, Artificial Intelligence, 26:pp. 251-321, 1985.

[3] P. Lima, L. Custódio, “Artificial Intelligence and Systems Theory Applied to Cooperative Robots: the SocRob Project”, Actas do Encontro Científico do Robótica 2002 - Festival Nacional de Robótica, 2002.

[4] Michi Henning, Steve Vinoski “Advanced Corba Programing with C++”, Addison Wesley, 1999.

[5] Bjarne Stroustrup. “The C++ Programming Language” , Addison-Wesley, Reading, 2000.

[6] “Mobility Robot Integration Software User’s Guide”, iRobot Corp. 2000.

[7] H. Bruyninckx. “OROCOS: Design and Implementation af a Robot Control Software Framework". Proc. of IEEE ICRA 2002, April, 2002

[8] Y. Hur, I. Lee. “Distributed Simulation of Multi-Agent Hybrid Systems", IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), April 29-May 1, 2002

[9] K. Konolidge. “Saphira Robot Control Architecture Version 8.1.0", SRI International, April, 2002

[10] Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, Won Soo Kim, "CLARAty: An rchitecture for Reusable Robotic Software," SPIE Aerosense Conference, Orlando, Florida, April 2003

[11] R. Alur, A. Das, J. Esposito, R. Fierro, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, J. Spletzer, and C. J. Taylor, "A framework and architecture for multirobot coordination," in Proc. ISER00, Seventh International Symposium on Experimental Robotics, Honolulu, Hawaii, Dec. 2000.

[12] D. MacKenzie, R. Arkin, and J. Cameron, "Multiagent mission specification and execution," Autonomous Robots 4(1), pp. 29--52, 1997

[13] Emery, R., Balch, T., Bruce, J., Lenser, S., et al. "CMU Hammerheads Team Description," submitted for RoboCup-2000: Robot Soccer World Cup IV, Springer Verlag, 2001.

[14] P. Lima, R. Ventura, P. Aparcio, L. Custdio. "A Functional Architecture for a Team of Fully Autonomous Cooperative Robots", RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000.

[15] P. Lima, M. Isabel Ribeiro, Luis Custodio, Jose Santos-Victor, "The RESCUE Project - Cooperative Navigation for Rescue Robots", Proc. of ASER'03 - 1st International Workshop on Advances in Service Robotics, March 13-15, 2003 - Bardolino, Italy, 2003

[16] D.C. MacKenzie and R.C. Arkin. "Evaluating the Usability of Robot Programming Toolsets" The International Journal of Robotics Research, Vol. 17, No. 4, pp 381-401, 1998.

[17] A. Vale, M. I. Ribeiro. "Environment Mapping as a Topological Representation", 11th International Conference on Advanced Robotics, Coimbra, 2003.

[18] A. Vale, M. I. Ribeiro. "A Probabilistic Approach for the Localization of Mobile Robots in Topological Maps", Proc. of the 10th IEEE Mediterranean Conf. on Control and Automation, Lisboa 2002.