Abstract—This paper describes the DSAAR architecture, which aims the fast development and prototyping of multi-robot systems and it is based on Linux inter-process communication (IPC) mechanisms and multi-agent Jade software platform. This architecture is of special interest in domains where several heterogeneous actors have to interact both with and without real-time constraints. One of these domains is humanitarian demining, where technology transfer is also a requirement.

I. INTRODUCTION

The motivation for the development of the DSAAR architecture is the current handcraft approach to robotics. Nowadays there is a lack of architectures that gather both computational and control models within the same framework. Robots are usually one-of-a-kind entities.

However, as robot technology becomes more and more mature, it becomes highly relevant to develop tools focusing on the fast development and prototyping of multi-robot systems. The DSAAR architecture intends to provide the engineer and/or researcher with a set of design models and implementation tools to respond to these requirements. At the current stage DSAAR can only provide a computational model (e.g. object-oriented, agent-based) without committing to any control paradigm (e.g. behaviour-based, deliberative). In the future it is expected to have some tools to instantiate DSAAR according to a set of given control models. Hence, the paper will focus on the computational model behind the architecture. Furthermore, it is shown that the computational model fits easily with the most relevant control models applied in mobile robots.

DSAAR is implemented using the multi-agent paradigm and intends to be generic enough to handle any control model, like Behaviour-Based ones (e.g. [1], [2] and Hybrid ones (e.g. [3], [4]).

DSAAR architecture is based on Linux inter-process communication (IPC) mechanisms and multi-agent Jade [5], [6] software platform. The former allows interoperability between independent processes running within the same machine, whereas the latter extend this concept to multi-agents that can spread through all over the internet. The success measure of this architecture should be specified in terms of prototyping speed and code reusability.

II. MOTIVATION: MULTI-ROBOT SYSTEM FOR HUMANITARIAN DEMINING

The DSAAR architecture development is part of the AMI-02 project, which aims the development of a multi-robot system for humanitarian demining (refer to [7] for a more detailed information about the project).

There are a set of characteristics about the humanitarian demining domain that makes it suitable for multi-agent approaches (see [8] for a study on this issue) and a multi-agent architecture was developed for this purpose, as illustrated in figure 1.

The architecture is composed of three layers, in which the physical agents layer refers to the actual robots. The software agents layer is responsible for product management (i.e. landmine detection sensors output and higher abstract data), configure/manage the whole system, and to create images of robot’s world perspective so as to allow operators to observe what is happening in the field. Finally, the human agents layer is populated with key personnel involved in the operations, which interact with the system (i.e. software and physical agents) in timely fashion.

Robots in the physical agents layer must be endowed with a certain degree of autonomy, so as to guarantee at least safe navigation capabilities and payload sensory information management. The architecture must be flexible enough to allow the addition and removal of robots and software agents as required. Hence, there is an actual need for distributed
new robots or robot types are required, users access to the architectural basis that allow developers to scale the system tightly coupled processes. Moreover, there is the need for an multi-agent systems support, along with a need for hardware tightly coupled processes. Moreover, there is the need for an architectural basis that allow developers to scale the system as new robots or robot types are required, users access to the system, etc.

III. PHYSICAL AGENTS LAYER COMPUTATIONAL MODEL

Figure 2 illustrates the two main software layers encompassing robots’ control system, the social ability layer and the individual control layer. In short, the former is composed of a set of agents responsible for defining the robot’s behaviour according to the robotic and human team intents. Note that these agents are not those represented in the software agents layer in figure 1, which are remote to the robot. On the other hand, the individual control layer aggregates a set of computational units whose scope is limited to the robot itself, and that are responsible for guaranteeing robot navigation and hardware resources management.

A. Social Ability Layer

The social ability layer is based on the Jade multi-agent platform, which is fully written in Java. Jade handles inter-agent communication in a transparent way, whether agents are in the same or in different machines. Communication among agents follows a well specified ontology. DSAAR defines for this layer the following roles:

- Implementation of collective behaviours, i.e. behaviours requiring interactions between different physical agents (i.e. robots);
- Local counterpart (i.e. interface) of the software agents layer represented in figure 1;
- Fast prototyping of robot control behaviours; notice that Java may induce considerable latency in cycle times and so control behaviours in Jade have to be considered in a very carefull and temporary way.

As previously referred, agents can be located in any machine, can even be moved from one machine to another. This allows a fast reconfiguration and adaptability of the system. For instance, agents can be moved to the machine as the control model requires. This also allows to have agents distributed as computational resources become available, distributing the load. In addition, agents can be launched in different machines (e.g. in a PDA or in a remote internet connection) and connected to a specific robot on the fly via local agents.

Hence, Jade makes DSAAR a highly versatile architecture. However, Java agents can not implement robot control algorithms. This is delegated to the individual control layer. In fact, a sockets-based API (Application Programming Interface) has been developed to allow the interaction between Jade agents and Linux processes, which are running at the individual control layer. This API is responsible for translating Java objects into C structures and vice-versa.

Please note that processes can be considered agents in a control model perspective, depending on their role. It has been chosen to distinguish computational units in both layers with the above lexical definition independent of their actual roles because the term agent is employed naturally in Jade and processes in Linux. At the control model the nomenclature can be altered according to a conceptualisation, which is not necessarily bound to the implementation details.

B. Individual Control Layer

At the individual control layer the system developer is endowed with a set of APIs and design guidelines to implement linux-based processes. With this API, the developer can implement processes with one or more of the following roles, depending on the control model:

- **Sensor Process.** A process responsible for fetching sensory data, pre-process it, and send the results to other process(es) via an IPC (Inter-Process Communication) mechanism (e.g. Shared Memory, Message Queues). In other words, the output message of such a process is a percept, which can be piled up in a Message Queue or stored in a shared memory associated to a time stamp.
- **Actuator Process.** A process responsible for acting in actuators according to incoming IPC messages, produced by other processes;
- **Behaviour Process.** A process responsible for achieving a particular goal of the robot according to incoming data from other processes (e.g. sensory or other behaviours) and forwarding data to other processes (e.g. actuators). This is responsible for the implementation of the robot’s control logic. In particular a behaviour process can be reactive, deliberative, or hybrid. The granularity of the semantics associated to each process is left to the control model and no assumptions are made at the computational model level. Nevertheless, the provided computational semantics is already driven by well established control model concepts, like behaviours and precepts.
- **Interface Process.** A process responsible for interfacing different systems, like Jade agents and Linux-Based processes. These processes are really important to integrate legacy systems, ad-hoc developments, and even software implemented in different hardware (e.g. interfacing two PCs through a serial port), which may be useful
to fulfil real-time constraints.

Figure 3 illustrates the main set of modules composing a process. The following sections describe each of these blocks in more detail.

1) **Process Logic**: This module is responsible for implementing all the logic corresponding to the process behaviour. That is to say that the process logic can be a sensory-motor module, a deliberative module, a behaviour coordination node, a sensory input signal processing algorithm, etc.

Therefore, the process logic definition is to be defined according to the control model, whereas the computational model is concerned about the way modules interact among themselves. The internal structure of the process logic is not covered in this paper.

2) **Linux Signals**: Linux already implements several mechanisms for process control. One of those mechanisms is 'signals'. Signals are sent to processes in order to control their activity (e.g., putting them to sleep, killing them). DSAAR architecture includes this feature as a convenient way of manipulating processes activity.

Linux signals allow the manipulation of process activity in a blind fashion; i.e., processes are not necessarily aware that they have less priority. Explicit process activity can be implemented in DSAAR, as it will be described later on.

3) **Inter-Process Interface**: Processes interact among themselves via IPC (Inter-Process Communication) Linux mechanisms. The Inter-Process Interface is responsible for abstracting all possible mechanisms under the same API. The IPC clients (message receivers) and servers (message senders) are specified in a configuration file. Hence, processes do not need to know who are the clients and servers at compilation time, they only have to know the communication semantics, i.e. message structures. Then, according to the mission at hand the configuration file is changed and processes redirect their outputs and inputs accordingly. Moreover, since messages are well defined one can change the processes generating messages as long as they respect the common semantics.

The following mechanisms are considered in DSAAR:

- **Shared Memory**: Of special interest for large data blocks transfer. The absence of a built-in "event" generating mechanism, requires from processes to poll for new messages. Linux semaphore mechanisms can be employed to ease this interaction;

- **Message Queues**: Messages are described as C structures, discarding the need for string parsings, as in the previous and subsequent cases. Messages trigger events and so processes do not need further synchronisation or polling mechanisms. It is possible to send messages to different clients by specifying the type of the message. However, there is a limited number of messages that can be flowing simultaneously.

- **FIFO**: Similar to Linux pipes, i.e. special files where processes can write and read messages. This mechanism also requires parsing and special care with messages with different sizes.

- **Sockets**: Sockets are of special use to interact with Java processes. Drawbacks related to string based communication.

Currently, a generic message is defined as a C structure which can then be directly sent to a Message Queue or translated into a string and then sent to a FIFO, a Socket or whatever. In addition, a set of functions to manipulate the C structure are provided in order to build up a complete API. Process developers can then make use of this API to send/receive messages to/from other processes via any IPC mechanism in a transparent way.

The goal of the architecture is to allow the developer to focus on the process logic. To do that, the designer only has to take into account that the process will require a set of inputs and outputs, through which messages flow. Thus, both the sender and the receiver must only know how messages are built.

For each message flowing in the system there has to be two corresponding xxx_msg.h and xxx_msg.c files. These files publish the following two major entities:

- A C structure whose fields specify the message body (e.g. heading), the time-stamp of the message, and in some cases the identification of the target client. This structure is to be filled in by the sender and read by the receiver.

- Two functions to convert a string into a structure and vice-versa.

The second function is required because there is no automatic way of translating a C structure instance into a string.

Message senders and receivers only have to include these files so as to abstract the messages' transmission medium. The medium through which a message is sent is not set in the referred files nor in the process logic. In fact the transmission medium (e.g. message queues) is set along the message recipient in a configuration file (see above explanations about this issue).

Figure 4 illustrates the internal structure of the implemented API. As it is possible to depict from the figure, the structural API (i.e. ipc.c) handles messages without being aware of their
structure (i.e. semantics). This API handles *void* messages. Then, the message customised API (i.e. *xxexecmsg.c*) provides the process with message semantics via the entities presented above. The control logic can interact with both. Preferentially, if a customised message API exists, then it should be used.

The structural API (i.e. *ipc.c*) is mainly composed of functions to manage IPC shared resources (e.g. creation) and to send and receive messages. These latter features depend on the IPC type (e.g. shared memory, message queue), the client identification, the IPC resource Key (i.e. unique ID), and the message itself. If the type is message queue then the message is the structure specifying it; otherwise the message is a string describing it, which has been translated by a customised API (i.e. *xxexecmsg.c*).

4) **Process Manager:** The process manager is thread internal to each process. The process manager is responsible for making the process compliant with DSAAR. Other processes can send dedicated IPC messages to the process manager (i.e. *modulate* type messages) to:

- **Allocate Memory.** This message asks the process manager to allocate and register all required IPC resources.
- **Start/Stop/Resume the process.** Self explanatory. This is different from sending a Linux signal to perform these operations. In this case the process logic can be stopped while other activities still running, like logging.
- **Change cycle time.** Changing cycle time is different from changing priority. When one changes priorities via the *renice* command the process runs slowly than others. Cycle time refers to the time spent without doing nothing before starting a new cycle, independent of the processing “speed”.
- **Inhibit IPC input messages.** Inhibition of messages can be used for both debugging and as a part of the control model.
- **Suppress IPC output messages.** Suppression of messages can be used for both debugging and as a part of the control model.
- **Others** ...

Thus, the process manager, among other functions, is responsible for maintaining an internal structure with the info received by the *modulate* messages. Then, the process logic shall comply with these modulation signals, like sleeping after a cycle so as to cope with the cycle time requirements.

This modulation mechanism is generic enough to allow:

- Synchronisation of several processes according to a general intent;
- Modulation of the activity of each process according to some events (either exteroceptive or proprioceptive).

Notice that *modulate* type messages are received and sent through the IPC DSAAR API, as any other message; hence, a process output can easily be configured to modulate other processes, which is a very important feature. It is common in robotic architectures to have processes modulated according to the current world and internal state. For instance, computational load of certain perceptual processes can be reduced according to some heuristics about the environment.

The process manager also provides the process logic with some API calls in order to read the configuration files, which describe who are the IPC clients and servers the process has to interact with.

5) **Hardware Interface:** The hardware interface is a collection of device drivers that allow linux-based processes to interact directly with real or simulated hardware (i.e. sensors and actuators). These device drivers are to be implemented, mainly, in C language.

IV. CONCLUSIONS AND FUTURE WORK

This paper presented the preliminary developments of the DSAAR architecture, which aims to be a flexible, scalable, and distributed multi-agent architecture for the development of multi-robot systems.

Some system specific processes and agents are used to build up a robot operating system. Thus, the control system designer is provided with a set of tools to speed up prototyping and incremental development.

As future work we intend to implement the remaining parts of the architecture and to create a complete robot operating system, with development tools, and a set of scenarios.

REFERENCES


