PRODUCTIVITY IN ROBOTICS SOFTWARE BY USING JAVA AND REAL-TIME JAVA

Diego Alonso Cáceres, dalonso@um.es *
Luis Manuel Tomás Balibrea, lmtomas@um.es **
Humberto Martínez Barberá humberto@um.es *

* *Department of Communications and Information Engineering, University of Murcia, 30071 Murcia, Spain*
** *Department of Company Organization and Finances, University of Murcia, 30071, Murcia, Spain*

Abstract: This paper presents a framework for designing, simulating, monitoring, and controlling a robot, which is aimed to the development of autonomous robots and is based on Java. By using this Java based framework it is possible to reuse a great amount of already done code when a new robotics systems is being developed, while still being able to use it in a great variety of platforms. On the other hand, the use of autonomous robots dramatically decreases the need for reprogramming the robot and modifying the environment. In the scope of this framework, the paper addresses the issues concerning the specification and satisfaction of timeliness constraints in order for the controlability of the robot can be improved.

Keywords: Mobile robotics, Java-based robots, Flexible AGV

1. INTRODUCTION

Every day, more and more industries are having their production lines automated. They are using each time more and more Automated Guided Vehicles (AGV) for daily duty tasks such as the internal movement of raw materials. The use of these robotics systems has increased during the last years and will surely suffer a sharply increase in the near future. But the main problems with these systems lie on their inability to adapt to a changing environment, the fact that new features can't be easily added to the robot program (such as a new algorithm or a new sensor) and that the software is usually designed *ad hoc*, specifically for the application, and that little or no code at all can be reused from one application to another.

But, is it the robot's fault?. No, the problem relies in that the software is conditioned by the hardware (mainly sensors and control systems) of the AGV. But now that the hardware is no longer a problem (CPUs are fast enough and good sensors exist; their price is another history), the only problem that lasts is the one related to the software. So a new framework for designing the control software for a robotics platform was started. It was clear what the platform should offer: it should be able to control, simulate, do the monitoring and teleoperation of a robot. It should be modular and allow the integration of new modules to the different systems of the framework without the need of major reprogramming. Since another goal of the design was to be able to execute it in a great variety of robotics platforms, Java was selected as the programming language, because it provides a portable, interpreted, high performance, object oriented programming language and supporting run-time environment for

a great variety of hardware architecture. And the Java implementation makes a simple process the integration of human interfaces in web based scenarios. The framework was called ThinkingCap–II (TC–II) (Martínez, 2001).

But it isn't the only one framework. There are many more frameworks dedicated or that can be used to make the design of the software to build up an AGV system, as for example the The Real-Time Control Systems (RCS) Library [1] developed by the Intelligent Systems Division of the National Institute of Standards and Technology, or the Open Robot Control Software (OROCOS) Proyect [2], but they aren't Java based robotics systems and they all run on top of Linux, so there's a bound to the operating system that can be use. Currently, there's only one another Java framework called TeamBots [3], which is being developed at Carnegie Mellon University.

The Real-Time Control Systems (RCS) Library is a software library that can be used to implement complex hierarchical and distributed control systems. The typical structure of a controller built using RCS consists of interdependent computing nodes, organized in a hierarchical structure, and (possibly) distributed over different (and possibly incompatible) platforms. It also provides a set of communication tools, so the computing nodes can still be connected to each other while running independently. It's a very regular architecture, because all nodes are composed of the same modules. It's a library intended for general real-time applications and it only provides the basic skeleton and communication facilities, so all algorithms have to be programmed and tested. Besides, no simulator nor teleoperation modules are built in it.

Orocos' primary aim is to provide a real-time software framework for robot control. The framework proposes some philosophies which are forced upon the user by design of the interfaces because Orocos aims to provide common interfaces for designers. In its current implementation, real-time execution of the program is provided by inserting kernel modules into the kernel, so it currently can only be executed under RTAI, Linux user-space and partially under RTLinux. It's major drawbacks: it's a very restrictive framework for robot design and it's conditioned by future changes on the way of programming modules.

TeamBots is a Java based collection of application programs and Java packages for multiagent mobile robotics research. The simulation environment is entirely written in Java. At present, TeamBots will run only on the Nomadic Technologies' Nomad 150 robot. TeamBots supports prototyping, simulation and execution of multirobot control systems. Robot control systems developed in TeamBots can run in simulation using the simulation application, and on mobile robots using the robot execution environment, but it doesn't has support (yet) for real-time.

[1] *http://www.isd.mel.nist.gov/projects/rcs/*
[2] *http://www.orocos.org*
[3] *http://www.teambots.org*

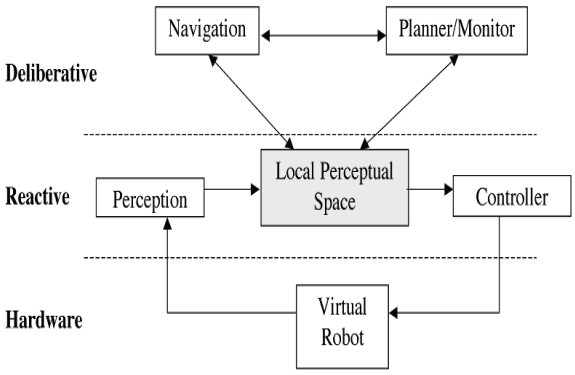


Figure 1. TC–II software layers

2. INTRODUCING THE THINKINGCAP-II FRAMEWORK

ThinkingCap–II (Martínez-Barberá and Gómez-Skarmeta, 2002) is a framework for developing mobile robot applications, and it is the extended combination of the ThinkingCap (Saffiotti *et al.*, 2001) and BGA (Gómez-Skarmeta *et al.*, 1999) architectures. The framework consists on a reference cognitive architecture (largely based on ThinkingCap) that serves as a guide for making the functional decomposition of a robotics system, a software architecture (partially based on BGA) that allows a uniform and reusable way of organizing software components for robotics applications, and a communication infrastructure that allows software modules to communicate in a common way independently of whether they are local or remote. Figure 1 shows the different layers that forms the structure of the framework and the relation between the different modules.

2.1 Functional architecture

The functional architecture is based on ThinkingCap. It is a two layers architecture for controlling mobile robots: one layer for reactive processes and the other for deliberative processes. The modules group the different functionalities present in typical mobile robotics systems (navigation, perception, control and planning), in which sensing and acting are a must. An important role is played by a centralized data structure called *Local Perceptual Space* (LPS). It is a geometrically consistent robot centric space which maintains a coherent model of the local environment of the robot, taking into

account the *a priori* information (map) and the currently perceived information (sensors).

The *Virtual Robot* module provides an abstract interface to the sensorial and motor functionalities of the robot, effectively hiding the hardware components. The *Perception* module receives sensor data from the Virtual Robot and applies a certain set of perceptual routines and intelligent sensor fusion. The *Controller* module closes the control loop by taking as input the information included in the LPS and generating as output robot control values. It is typically implemented as a library of fuzzy behaviours for navigation, like obstacle avoidance, wall following, and door crossing. These three modules form the reactive layer.

The *Navigation* module is in charge of modelling the environment by using both the LPS and an *a priori* world model (if available). It typically includes a series of map-building, localization and path-planning algorithms, like fuzzy grid maps, fuzzy segments maps, topological maps, Kalman filters (Kalman, 1960), and A* and D* planners. Finally, the *Planner/Monitor* module generates and supervises plans that are needed to solve the robot task. These two modules form the deliberation layer.

2.2 Software architecture

The run-time characteristics of the system can be specified and customized by the use of configuration files. The kernel supports three different types of configuration files, and contains methods to parse and check them: Architecture Definition File ADF (specifies which modules are to be run, in which CPUs they will be running on, which type of communication and process synchronization mechanism they will be using), Robot Description File RDF (specifies the different parameters related to a given robot, like sensor number, types, position and orientation, platform kinematics type and parameters) and World Description File WDF(specifies the *a priori* knowledge of the robot environment,like walls, rooms, corridors, landmarks, areas, waypoints, etc, and can be left empty if no *a priori* information exists).

The TC–II framework includes a *Simulator* module that can realistically simulate multi-reflection range sensors, faulty sensors, laser based localization sensors and different kinematics platforms (differential, tricycle and Ackerman drive). For all of these, different error models can be selected. The environment model is specified with an ADF. The switching between a real robot and a simulated one is as easy as changing the class name of the Virtual Robot section in the ADF.

Due to the nature of the functional and software architecture, information must be exchanged between the different modules. Moreover, the system allows running the different modules in a distributed way, making information exchange crucial. The solution chosen was to implement a shared blackboard in a similar way as in a Linda system (Gelernter, 1985). As in typical blackboard systems, each module reads information from the blackboard, processes it, and then writes the corresponding results. Besides this, it can also work as an event driven blackboard. In this way, each module registers into the blackboard which kind of data it desires to receive. When new data of such type is available it is directly sent to the module.

3. USING REAL-TIME JAVA

First of all, two questions arise: what is a real-time system and what advantages could it add to the framework? The canonical definition of a real-time system (from Donald Gillies), is the following: "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timeliness constraints of the system are not met, system failure is said to have occurred". One important question, that has to be clarified, is that the adjective "real-time" is not necessarily synonymous with "fast", that is, it is not the latency of the response *per se* that is at issue (it could be of the order of seconds), but the fact that a bounded latency, which has to be sufficient to solve the problem at hand, is guaranteed by the system.

Application like robot control requires a strict control of time. It's very important to assure that no process locks the CPU for a long time and that every process is executed according to its priority and frequency. Processes like the one in charge of doing the control of the movement of the robot or the update of the readings of the sensors have to be regularly executed because they are critical for the good operation of the system.

The Real-Time for Java Expert Group was chartered under the Java Community Process to produce a specification for additions to the Java platform to enable Java programs to be used for real-time applications: the Real-Time Specification for Java[4] (G., 2000). By the time this paper was written, the only implementation of the Real-Time Specification was the one developed by TimeSys[5] on top of their version of the Linux

[4] *http://www.rtj.org*
[5] *http://www.timesys.com*

Kernel. The Java Real–Time API enables the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints. This API provides the needed tools to develop applications with these timeliness constraints. Specifically, it provides enhancements in the following areas:

Thread Scheduling and Dispatching. The Java Real-Time API comes with a predefined scheduler policy, which is priority-based, preemptive and with at least 28 unique priorities, but it also permits the definition of new scheduling policies. Two new kind of threads are defined , which can be scheduled and dispatched by the scheduler, according to parameters such as its priority, period, cost and deadline.

Time. The API provides new classes to deal with time issues, such as time objects with nanoseconds precision and timer objects that trigger when a certain amount of time has elapsed.

Memory Management. Memory management algorithms (garbage collector, GC) can continue being used but the effect on the execution time, preemption and dispatching of the other threads has to be characterized. It implies that a Realtime thread can have a higher priority than the garbage collector.

Synchronization and Resource Sharing. One of the biggest problems related to real-time systems is the *priority inversion* problem. To deal with this problem, the two most common mechanisms of avoidance have been programmed: priority inheritance and priority ceiling, by using the keyword *synchronized*, but new ones can be programmed and added. It has also been implemented a mechanism of synchronization between Realtime thread and normal Java threads, by means of queues, in order to assure that the time restrictions are fulfilled.

Asynchronous Event Handling. Just because the real world is asynchronous, the Real-Time API provides the needed mechanisms to deal with it. Events that can happen at any time are represented by objects linked to the logic that has to be executed when the event happens. The logic associated to these asynchronous events can be then scheduled (it has its own priority) and dispatched by any scheduler.

Asynchronous Transfer of Control. Sometimes the real world changes so drastically that the current point of execution is no longer valid and should be transferred to another location. However, this change can be only possible if the program is specifically written to allow this change. One of the major advantages of this transfer is that threads can be stopped in a safe way by means of this mechanism.

Physical Memory Access. Although is not directly related to real-time issues, sometimes it is interesting to have a low level access to the hardware physical memory.

4. INTEGRATING REAL-TIME JAVA TECHNOLOGY AND THINKINGCAP-II

The last part to do is modifying the framework to adapt it to use the Real-Time characteristics of the new API. As shown in section 2, the framework is based on three layers, each in charge of a different task, but all inter-dependent: the cognitive layer, the execution layer and the communication layer. Since the execution layer is the lowest level layer and the one that starts the application, it is, obviously, the one that has to be modified.

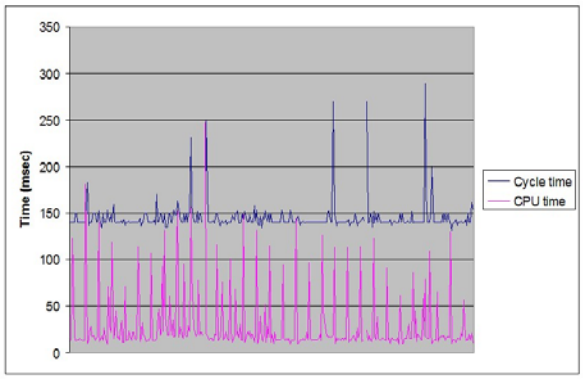
The execution layer provides the following services and features: abstraction from the hardware layer using a virtual robot, a way to execute all the modules depending on the ADF, a way to customize the modules (the qualified class to be instantiated, the parameters and other characteristics are chosen in execution time) and a link to the communications layer accessing Linda blackboard and servicing Linda events. New parameters have been added to the ADF to specify whether the different modules (as, for example, the virtual robot or the planner) should use or not the Real-Time API and the timeliness constraints of the real-time modules. Obviously, the core of the execution layer has also been modified, extending the thread-base class to allow the specification of both kind of threads (normal or real-time threads), depending on the specification loaded from the ADF, and to start them with the specified timeliness constraints.

5. EXPERIMENTS AND RESULTS

Several tests have been done to check the operation of the framework in real robots. Among all these tests, only the ones related to the time behavior of the framework are presented. All the tests have been run over the same hardware platform: an industrial CPU card, TH-512 All-In-One Little Board, based on a Transmeta Crusoe processor working at 500 MHz and with 128 MB of RAM memory, called Mungo. These tests show the need to add real-time characteristics to the robot control software to improve the global behavior and response of the system.

5.1 First test. TC–II time response

This first test was ran over the Mungo board with Red Hat Linux 7.1. Figure 2 shows the



	Cycle Time	CPU Time
Mean	146,636971	46,23385301
Std. Dev.	17,45580045	31,97642985
Minimum	134	12
Maximum	328	239

Figure 2. TC–II control cycle under Red Hat Linux 7.1

time results of the execution of the TC–II control modules over the hardware that is being used on a normal robot. As can be seen it the figure, it's mandatory to stabilize the frequency of execution of the control cycle in the robot to assure a better overall response and behavior of the robot.

5.2 Second test. Using a real-time OS with TC–II

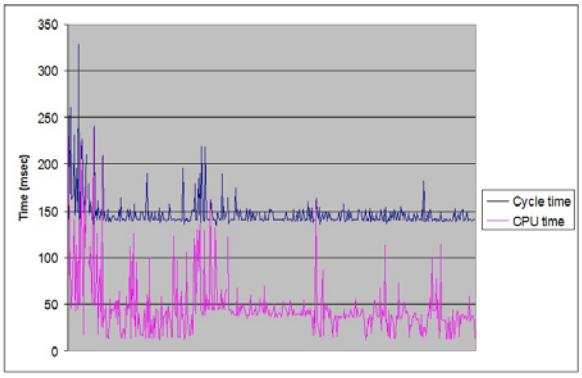
This second test tried to check out if it would be just enough to run the same program, the TC–II robot control modules, with no modification to the framework, over a real-time operating system to get the desired time behavior, or, at least, a slightly improvement over the first test results. To make this test the TimeSys 3.1.214 Linux Kernel was installed and configured on the same Mungo board and the same robot control program was then ran. Seeing the results of this second test (figure 3), it is observed that the execution of the Java Virtual Machine over a real-time operating system such as TimeSys's provides *no improvement* on the time behavior of the application, and the desired timeliness constraints couldn't be achieved.

5.3 Third test. Applying the Real-Time API

The last test tries to check out if the results of modifying the framework so that it can use the real-time characteristics of the new Java API would be satisfactory. Specifically, it is tried to find out if the timeliness constraints imposed to each thread executed by the robot core CPU will be fulfilled. This test involves the program of an application in charge of simulating the CPU load when executing the TC–II control modules on a real robot. Of course, and just like it was done in the previous tests, this program was ran on the same Mungo board. The Mungo was running the TimeSys Linux Kernel and had installed the TimeSys's Java Real–Time implementation. The TC–II simulator was done trying to be as similar to the real program as possible. Thus, the simulator consists in three kinds of execution threads:

- Two **synchronous** threads, which simulate the two tasks which are periodically done by the control module: the control cycle of the robot (which involves the calculation of the control action and the update of the sensor readings) and the planner module.The timeliness constraints for these two threads are the same: they should be executed by the CPU every 150 msec., and both threads have been found to take around 20 msec. to complete their cycle. Of course, these two threads have both the same priority, which is the highest of all threads simulated.
- Three **asynchronous** threads, which try to simulate operations such as receiving/sending information from/to data sockets or serial ports.
- One thread that have to be executed when none of the previous threads are in execution and, thus, have the lowest priority. This thread try to simulate all the tasks which are not critical for a good system operation. The cost of the execution of this threads has been estimated in around 10 msec.

The results of the execution of the TC–II simulator are shown in figure 4. As can be seen in the figure, the results were right the expected ones. Every thread is executed according to its priority and its time characteristics, which are both under control of the system's designer. Moreover, the two synchronous threads, which are the most



	Cycle Time	CPU Time
Mean	144,654762	28,625
Std. Dev.	16,0422904	33,5090431
Minimum	132	9
Maximum	290	247

Figure 3. TC–II control cycle under Timesys Linux 3.1.214

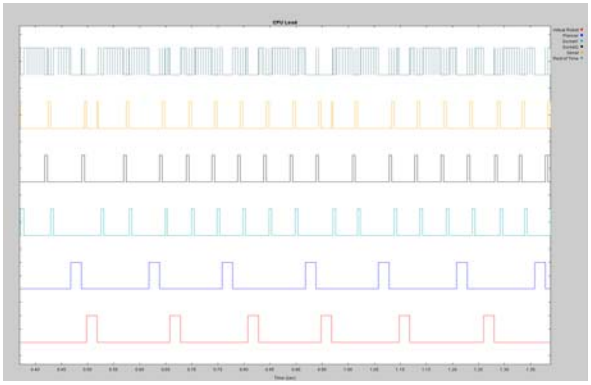


Figure 4. TC–II CPU load simulator with full real-time support

important ones, are always executed at the right time and always every 150 msec. This happened in every test that was run and will surely make a great improvement in the overall behavior and response of the robots.

6. CONCLUSIONS AND FUTURE WORK

This papers has presented a framework to deal with all the issues related to the cycle of life of an AGV, from design to upgrade once it's working, the ThinkingCap–II. This framework simplifies the design of new robots from scratch and the update of existing control software because it:

- Provides a template to follow when designing the control software of a new robot.
- Reuses practically 100% of the code that has already be done.
- Is very modular, so new functionalities can be added without the need to make major changes to the framework.
- Only robot/application specific issues, such as, for example, the size of the robot, the sensors and their position or the task the AGV is going to do, has to be programmed.
- Very configurable thanks to use of description files (ADF, RDF and WDF).

This framework has been tested with several robots, ranging from hand-made robots for study and development to more serious robots, as can be a forklift used in an agricultural company, and it has shown good capabilities to serve as an abstract guideline to organize the software which has to be run in a module robot. Moreover, is simple enough to help understand how a given robot works.

Trying to improve the design and performance of the framework, an effort is being made in order to add real-time technology support to the TC–II because of the advantages it would provide to the AGVs, as commented in section 3. To do so, the first implementation of the real-time specification for Java, made by TimeSys for their Linux GPL, was used and tested to find out if it was capable of fulfilling the timeliness constraints imposed by the framework.

ACKNOWLEDGEMENTS

This work has been supported by ART-11 contract "Automatización de Carretilla Paletizadora", El Dulze S.A., and CICYT project TIC 2001-0245-C02-01, Ministerio de Ciencia y Tecnología, and by the proyect "Integración de Tecnologías Inalámbricas", Ministerio de Fomento. Diego Alonso Cáceres has been supported by *Consejería de Trabajo y Política Social de la Región de Murcia and the European Social Fund* through the convocatory of grants by *Fundación Séneca.*

REFERENCES

G., Bollella (2000). *The Real-Time Specification for Java*. first ed.. Addison-Wesley Pub Co.

Gelernter, D. (1985). Generative communication in linda. *ACM Trans. on Programming Languages and Systems* **17**(1), 80–112.

Gómez-Skarmeta, A.F., H. Martínez-Barberá and M. Sánchez (1999). A fuzzy logic based language to model autonomous mobile robots. In: *Procs. of the Eighth IEEE Intl. Conf. on Fuzzy Systems*. Seoul, Korea. pp. 550–555.

Kalman, R.E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Trans. ASME, J. Basic Engineering* **82**, 34–45.

Martínez-Barberá, H. and A.F. Gómez-Skarmeta (2002). A framework for defining and learning fuzzy behaviours for autonomous mobile robots. *Intl. J. of Intelligent Systems* **17**(1), 1–20.

Martínez, H. (2001). A Distributed Architecture for Intelligent Control in Autonomous Mobile Robots. PhD thesis. Dept. of Information and Communications Engineering, University of Murcia, Spain.

Saffiotti, A., , M. Bomanand, P. Buschka, P. Davidsson, S. Johansson and S. Wasik (2001). Team sweden. In: *RoboCup 2000: Robot Soccer World Cup IV* (P. Stone, T. Balch and G. Kraetzschmar, Eds.). pp. 643–646. Springer.