Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.

This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

ELSEVIER

Robotics and Computer-Integrated Manufacturing 24 (2008) 150–166

Robotics and Computer-Integrated Manufacturing

www.elsevier.com/locate/rcim

A software engineering approach for the development of heterogeneous robotic applications ☆

Juan-Antonio Fernández-Madrigal*, Cipriano Galindo, Javier González, Elena Cruz-Martín, Ana Cruz-Martín

System Engineering and Automation Department, University of Málaga, Complejo Tecnológico, Campus Teatinos, 29071 Málaga, Spain

Received 1 December 2005; received in revised form 21 July 2006; accepted 1 October 2006

Abstract

One of the most evident characteristics of robotic applications is heterogeneity: large robotic projects involve many different researchers with very different programming needs and areas of research, using a variety of hardware and software that must be integrated efficiently (i.e.: with a low development cost) to construct applications that satisfy not only classic robotic requirements (fault-tolerance, real-time specifications, intensive access to hardware, etc.) but also software engineering aspects (reusability, maintainability, etc.). Most existing solutions to this problem either do not deal with such heterogeneity or do not cover specific robotic needs. In this paper we propose a framework for the integration of heterogeneous robotic software through a software engineering approach: the BABEL development system, which is aimed to cover the main phases of the application lifecycle (design, implementation, testing, and maintainance) when unavoidable heterogeneity conditions are present. The capabilities of our system are shown by its support for designing and implementing diverse real robotic applications that use several programming languages (C, C++, JAVA), execution platforms (RT-operating systems, MS-Windows, no operating system at all), communication middleware (CORBA, TCP/IP, USB), and also a variety of hardware components (Personal Computers, microcontrollers, and a wide diversity of sensor and actuator devices in mobile robots and manipulator arms).
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Software engineering; Robotic software; Distributed programming

1. Introduction

A pervasive characteristic of robotic systems that has a great influence in developing high-quality applications is their heterogeneity. That is, robotic projects involve very different areas with very different needs: artificial intelligence, control systems, data acquisition, networking, etc., which requires the collaboration of very different people and the integration of a variety of software and hardware components. In spite of the difficulty of integrating this diversity with the lowest cost of development, it is also necessary to produce working, robust, and dependable applications. Solving all these requirements at once is an unavoidable problem as robotic projects increase in size and complexity, so much so that a bad (or inexistent) solution can lead to low-quality implementations, and/or to increase the duration of development up to an unacceptable point. When referring to the software aspects of this issue we will refer to it as the *robotic software integration problem* (RSIP).

Although the RSIP has been approached in some works in the last years, it has not been the subject of in-depth research in the robotics literature (maybe in the belief that it is a problem for the software engineering area); thus it has been dealt with only through partial solutions (some of them will be reviewed in Section 2) that in addition tend to consider it as a minor problem. On the other hand, software engineering reasearchers rarely have identified the particular heterogeneity aspects that make robotic software different.

☆This work was supported by the Spanish Government under research contract DPI2005-01391.

*Corresponding author. Tel.: +34 952 132892; fax: +34 952 133361.
E-mail address: jafma@ctima.uma.es (J.-A. Fernández-Madrigal).

0736-5845/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.rcim.2006.10.002

Author's personal copy

We have been dealing with the RSIP since several years ago, implementing and enhancing different solutions for a number of robots and applications (mainly mobile robots like the ones described in Section 3) [1,2,4,7]. This has led us to the proposal of a global framework for the problem, since we have found that it is very difficult for any partial approach to succeed. In addition, we have decided not to build up such framework from scratch, but to design it in a way that existing approaches (both commercial and non commercial) can fit. This has optimized our proposal by taking advantage of the best solution for each part, and also has allowed us to capture heterogeneity from the very beginning. The result, presented in this paper, is the BABEL development system: a bundle of specifications and tools for facilitating the design and implementation of robotic software applications in a clean and orderly fashion. BABEL is able to capture a high level of heterogeneity from the design of the application to its implementation, along with classical robotic requirements (fault-tolerance, real-time behavior, etc.).

BABEL takes from the Waterfall and the Iterative lifecycles [8] (see Fig. 1), that are two of the best known models for the software lifecycle, the main stages in which heterogeneity must be taken into account for a robotic application, namely: design, implementation, testing, and maintainance. Other stages are also of great importance, such as the analysis or validation phases. We are currently working on them for extending the BABEL capabilities. For the stages mentioned before, we have developed different tools: (i) the Aracne specification, based on *active objets*, which provides the basis for dealing with the RSIP in the design phase; (ii) the BABEL module designer (MD), a CASE tool which automatizes the production of heterogeneous implementations from Aracne designs; (iii) the BABEL execution manager (EM), which controls and sequences the execution of the different parts of the application even in distributed (and diverse) networked systems, and also retrieves information from testing; (iv) the BABEL Debugger (D), which allows the researchers to examine the results of execution in detail and test the satisfaction of real-time and scheduling requirements; and (v) the BABEL Development Site (DS), a web site (*babel.isa.uma.es/babel*) used as a repository of software components for the intensive reusing of code, which also includes some simple validation tools. The BABEL system as a whole has been used during the last years with important benefits for our research in other robotic areas [3,5]: it has reduced greatly the cost (time, effort) of developing new applications and has allowed our researchers to focus on the robotic problems they work on rather than on the integration of the components needed for those problems. We illustrate in this paper its capabilities with some real robots where it has been employed.

The structure of the paper is as follows. Section 2 presents some previous and related work. Section 3 describes a variety of heterogeneous robotic applications that will illustrate the use of our framework. Sections 4, 5, 6, 7, and 8 deal, respectively, with the tools and specifications of BABEL for the design, implementation, execution, debugging, and maintainance of robotic applications. The paper ends with a section in which some results on the benefits of BABEL are provided, as well as some conclusions and future work.

2. Previous and related work

During the last decades, a variety of tools and techniques aimed to address the RSIP have been proposed in the literature (although not under that denomination). Most of them provide meaningful insights into very specific domains or particular levels of abstraction of a robot application, but none of them deals with the whole problem, which is the main motivation of BABEL.

At a *high level* of abstraction, where the issue is to provide software frameworks for implementing intelligent decision procedures, tools like the MAESTRO language [9] or the temporal planning system called IxTeT [10] are relevant examples. On the other hand, at a *low level* of abstraction, the emphasis is on providing the programmer with a direct interface to the hardware of the robot, and several robotic real-time operating systems (Harmony OS [16], Chimera [17], etc.) have been proposed for that.

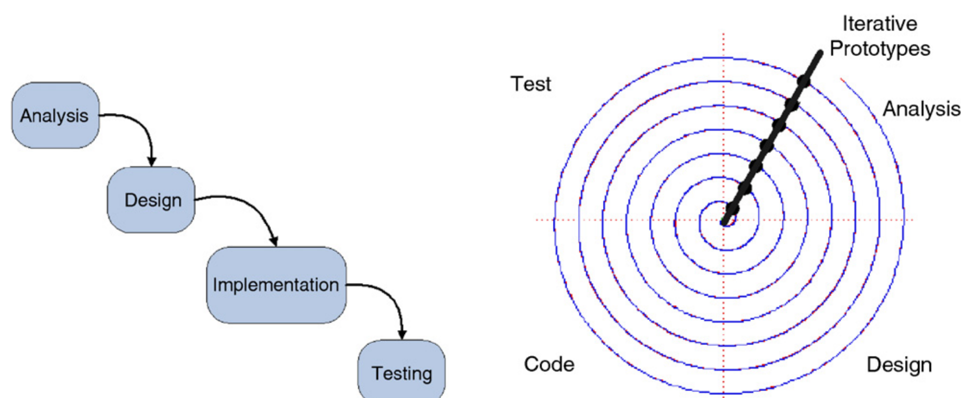

Fig. 1. Two well-known models for the lifecycle of a software application. On the left, the classical Waterfall lifecycle. On the right, the Iterative lifecycle. Both cover the development and use of the system in different ways, but exhibit similar phases.

Author's personal copy

At an *intermediate level*, which is the most relevant for our purposes, many approaches that enable the integration of functional components have been proposed: TCA [11], NASREM [12], some commercial packages such as CODE by Cimetrix [13], Constellation by RTI [14], etc. At this intermediate level, it is of special interest the $G_o^{en}M$ framework [15]. It provides programmers with a common framework for communicating and integrating distributed robot applications, while maintaining robustness, real-time response, and efficiency. However, it does not address comprehensively the RSIP through the whole software lifecycle. Among the free frameworks (non-commercial) there are also some works aimed to facilitate the implementation of robotic software efficiently, however none of them has been widely used in the industry (although OSACA, mentioned further on, has been developed with the participation of industrial companies). Rather, they are research approaches. An example is [18], where an agent and object-oriented based approach [19] is presented, but they do not focus on heterogeneity of robotic software, setting their approach specifically to some aspects of the functionality of modern hybrid robot architectures. Player/Stage [43] is a good example of an open source bunch of software tools that enable the control of robot and sensor devices (although it is not a complete framework since client-side software must be developed entirely by the user). Also, OROCOS [20] provides a free-cost, suitable set of pre-built libraries for machine tools and robotic arm control (mobile robotics is still under development). In the industrial robot arena, some open approaches are OSACA in Europe, OSEC in Japan, and OMAC in the US [21], although neither one has been widely adopted by industry. A more general perspective is present in CORBA [22], a middleware that establishes a standard view of underlying communication technologies among programming objects that can be coded in different languages and has some free implementations available, but it is not focused on the special characteristics of robotic applications.

Our work on the RSIP began with a partial solution, similar to $G_o^{en}M$, called NEXUS [1,4], that allowed programmers to easily integrate different modules of an application while maintaining specific robotic requirements. However, NEXUS did not deal comprehensively with different phases of the development lifecycle and only worked on a particular real-time operating system (LynxOS [36]) and programming language (C). NEXUS was the first step towards our BABEL Development System, which uses a software engineering philosophy that covers most of the phases of the lifecycle of the application. The basic idea underlying BABEL is not to provide a single, new approach for software integra tion (or alternatively, using one already existing), but to take advantage of most of the existing ones into a general framework, as described from Section 4 and on. Table 1 shows a comparison between BABEL and three other important frameworks, which illustrates relevant differences.

Table 1
Comparison between the BABEL framework and other relevant ones

Framework	Lifecycle phases covered	Automatic generation of code	OS limitations	Programming language limitations	Communication platform limitations	Real-time limitations
OROCOS	Implementation, testing.	No	Linux 2.6 currently. Complete implementation of the framework for other OSs.	C++	CAN, CORBA in progress	RTAI
OSACA	Implementation, testing.	No	Win32, VxWorks currently. Complete implementation of the framework for other OSs.	C++	TCP/IP	None
Player/stage	Implementation, testing only in simulation.	No	Linux/Solaris/BSD in the server side; any in the client side	Any in the client side	TCP/IP	The ones of the client side.
BABEL	Design, implementation, testing, maintainance.	Semiautomatic	Win32, LynxOS, JAVA VM, and no-OS currently. Only few modifications for further OSs.	C, C++, JAVA currently. Only few modifications for further languages.	ACE+TAO CORBA, TCP/IP, USB and Monolithic currently. Only few modifications for further platforms.	LynxOS RT and RTX support currently. Only few modifications for further platforms.

Notice that all the frameworks have limitations with respect to their support for different OSs, programming languages, communication platforms, and real-time facilities; however, extending BABEL with further support capabilities is easy since it is based on a well-defined design framework that enables heterogeneity. Also, the coverage of the software lifecycle is wider in our approach.

Author's personal copy

3. Heterogeneous robotic platforms for babel evaluation

One of the main benefits of using BABEL is that it enables the reduction of development cost when a diversity of robotic platforms is present. In this section we describe briefly some of our platforms, for which we have developed software applications using BABEL in the last years. In the next Sections (4 and on) we will describe the BABEL framework illustrating it with these platforms. Fig. 2 shows a general view of our mobile robotics laboratory with three of our robotic platforms. Fig. 3 shows details of an industrial manipulator arm that has also been programmed with our system. In the next section we describe with more detail these settings.

SENA (Fig. 2-middle) is a robotic wheelchair based on a commercial powered wheelchair Sunrise Powertec F40 [23] that has been equipped with an onboard computer and several sensors: a 180° SICK PLS radial laser scanner for mobile obstacle detection, environment map construction [24], and robot localization [25]; two ultrasonic Polaroid rotating sensors also located in front of the wheelchair and mounted on servos which enable them to scan a range of 180°; a ring of twelve infrared Sharp GP2D12 sensors placed around the robot to detect close obstacles; a CCD JAI CV-M300 B&W camera situated on a pan-tilt unit at a position similar to that of a standing-up person. It is also used to localize SENA [26]. The wheelchair is intended to reliably perform assistant tasks for mobility-impaired people in

Fig. 2. A view of three mobile robots in our research laboratory. From left to right: our mobile manipulator RAM-2, our assistant robotic wheelchair SENA, and our service robot SANCHO.

Fig. 3. Different views of the performer MK2 industrial manipulator arm. It is mounted on the RAM-2 mobile robot (as well as its controller-B), although it can also be used independently on the platform. In the figure you can see the main features and components of the robot, including the camera and collision sensors in the gripper.

Author's personal copy

indoor environments. Notice that the original wheelchair has undergone minimal modifications: we maintain the original controller and motors, although two encoders have been connected to the motors' axis for odometry. Sensorimotor hard real-time management is performed by a microcontroller ATMEGA 128 at 16 MHz connected to a laptop computer via USB. This microcontroller is in charge of managing the vehicle motors as well as some analog input signals like encoder information and infrared readings. The original joystick line has also been bypassed to be managed by the controller. However, we do not prevent the user to take the manual control of the vehicle, enabling her/him to disable the autonomous navigation of SENA at any time. SENA also accounts for an onboard laptop computer mounted on a retractable arm rest, which displays information to the driver, commands the microcontroller, and also has wireless communications capabilities to connect to remote servers or to the internet. Two small speakers and a bluetooth headset allow non-conventional interfacing with the user through a speech generation [27] and a voice recognition software [28].

SANCHO (Fig. 2-right) is a mobile robot intended to work within human environments as, for example, a conference or fair host. It is constructed upon a pioneer 3DX mobile base [29], on which different elements are placed. A structure has been devised to contain the sensorial system (which is analogous to the one of SENA), that is composed of a radial laser scanner, a set of 10 infrared sensors and a colour motorized camera (SONY EVI-D100P). The user-communication system entails a pair of speakers, a microphone, and a TFT color screen for displaying information. All elements of SANCHO are managed by an onboard laptop which is communicated to a remote station via wireless ethernet.

RAM-2 (Fig. 2-left) was constructed entirely from scratch. It is octagonal shaped, and its structure is aimed to bear a high load (including a manipulator arm). The locomotive system of RAM-2 is composed of two motor and two steering wheels managed by a DCX PC100 controller [30], while its sensorial system used for navigation entails a PLS 180° frontal radial laser scanner, a non-commercial Explorer 360° radial scanner placed on top of the vehicle, and a set of 14 infrared sensors placed around the vehicle at different heights. Sensors and actuators of RAM-2 are managed by an onboard industrial PC computer which can connect through ethernet to a remote station. RAM-2 was intended to autonomously manipulate objects, and thus it carries an industrial onboard robotic arm (described below), including its controller [31]. The arm controller is connected to the onboard computer through RS-232. Also, the latter receives information from a CCD camera attached to the wrist in order to assist in object manipulation.

The industrial manipulator arm PERFORMER MK2 (Fig. 3) is a commercial five-DOFs manipulator arm [44] with servo-controlled joints. The manipulator is managed by the Controller-B that executes programs written in the ACL language with real-time, inverse kinematics, input/output, and concurrency capabilities, and it is connected to an external PC through a conventional RS-232 link. We have attached to the robotic arm six infrared sensors placed at the arm gripper to detect objects and collisions, and a colour camera JAI 2060 [46]. Images from the camera are captured by a Meteor board card [47]. Our setting has been used for a variety of demonstrations, for example, to recognize and pick up small plastic pieces discriminating them by their colour.

4. Design of robotic applications in BABEL: the aracne specification

In this section we focus on the core of the BABEL development system: the Aracne specification that sets the guidelines for other tools. This specification covers the design phase of any robotic software application with heterogeneity needs.

The key point in Aracne is *flexibility*: it must be able to integrate under the same design both very dependable and barely (or not at all) dependable software, to support very different implementation and execution platforms, to provide facilities for the use of networking or not, to exhibit capabilities of designing both large and small systems, etc. For that purpose, the first mechanism that Aracne provides is a clear separation between the *portable* and *non-portable* parts of the design. We call *portable* to the part of the design that is independent on any particular implementation (for instance: an algorithm to manage a symbolic model of the robot's world should be implementable on different operating systems, hardware, etc., without substantial changes). On the other hand, the *non-portable design* captures the design aspects that are tied to a particular implementation (for instance, a piece of code for reading a particular sensor is tied to that hardware and surely to the operating system). A clear divide between portable and non-portable improves reusability and facilitates the integration of the diversity of components often present in a robotic system.

The Aracne specification consists of *entities* connected by relations (see the pseudo-UML diagram in Fig. 4). Such structure is general enough for supporting any type of robotic architecture (deliberative, reactive, hybrid, behavior-based) or application. The rest of the section describes Aracne entities and their relations within the specification.

4.1. The portable part of the design

Portable entities have been designed under an object-oriented (more precisely, *active objects*) perspective [19]. They are classified into two categories. On the one hand, *structural portable entities* are used for specifying the static structure of the application. In this category, the application is composed of *modules*, which provide *services* to other modules, maybe through a network. The second category is composed of *codification portable entities*, that

Author's personal copy

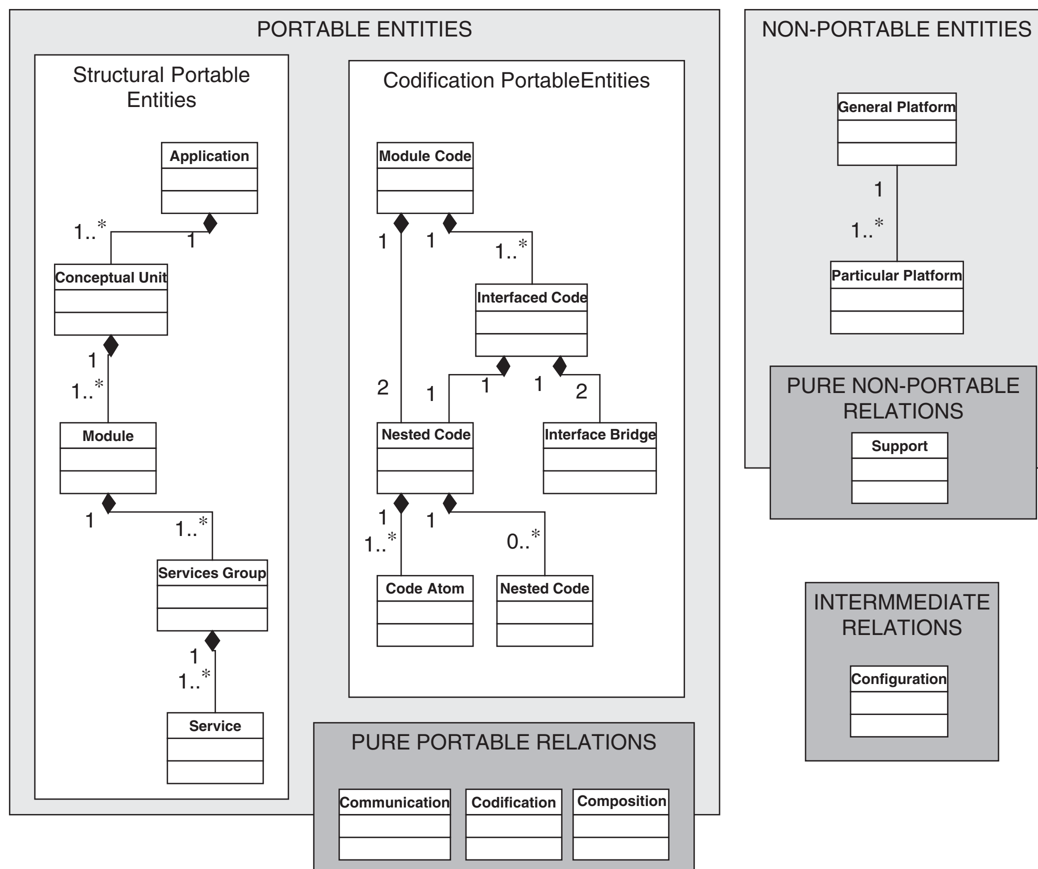

Fig. 4. The structure of any application designed with Aracne (for simplicity, it does not fit UML strictly). Structural portable entities serve to design the static and portable structure of the application. Codification portable entities serve to design the code (improving the reusing of code). Non-portable entities serve to design the part of the application tied to a particular hardware/software.

are still portable but closer to implementation. This category is used for specifying in some programming language or formalism the execution logic of structural portable entities.[1]

Structural portable entities: The structural portable entities are the *application*, the *module*, and the *service*.

The *application* consists of a set of modules. The *module* is the minimal, indivisible entity that can be executed on a certain machine (computer, microcontroller, robot controller, etc.). Notice that the codification design corresponding to a module can be portable or not (for example, a module can provide the functionality of the PERFORMER MK2 controller to the rest of the application, hiding its hardware details). The non-portable aspects can be conveniently isolated from the portable design by using some features of codification entities, described further on.

[1]Programmers may be surprised by the inclusion of codification (programming) into the *design* of the application. However, the especification of the logic of execution can be portable (assuming that a compiler/interpreter that produces executable implementations for that logic exists for any available CPU platform) as long as the non-portable parts of that code are carefully isolated. Our decision is intended to enable both the use of validation techniques on codification, and the reusability of a given logic for different implementations.

In our robotic applications, we have dealt with modules with both portable and non-portable codifications. Some of them are listed in Table 2. The modules marked with "low portability" have codifications that are tightly related to some specific software or hardware. However, their structural design is completely portable (that is, they provide to other modules public services with definitions that are not dependent on any platform). For example, the position estimator module for our mobile robots can reset odometry errors in two different ways: using a laser point matching algorithm [25], or by matching visual landmarks extracted by a camera mounted onto the robot [26], but the services it provides to the rest of modules are the same independently on the method used.

A module provides *services* to other modules (maybe to itself). The services of a module express its functionality. For reusability purposes, the part of the module that implements the logic associated to each service is kept private to the module.

As an active object of the application, a module also maintains an internal status that is not accessible from other modules, except through services. This status consists of data managed by appropriate initialization and termination logics and by the logics of services.

Author's personal copy

Table 2
Examples of some modules developed with BABEL for our robotic applications. Notice the variety (that is, the heterogeneity) present in these setups

Module	Description	Portability
PERFORMER MK2 controller	Interface module that controls the industrial manipulator arm. It also is able to read the status of the sensors attached to the end-effector. It runs on the external PC and communicates to the robot controller through RS-232 internally.	Low
SENA microcontroller module	Interface with motors, odometry, and I/O acquisition circuitry. Runs on the microcontroller of the SENA platform.	Low
RAM-2 Arm Motion	Interface with the RAM-2's arm motors. Runs on the onboard computer of the RAM-2 platform and manages the manipulator through a RS-232 connection to its controller.	Low
PLS laser manager	Interface with the frontal laser range sensor included in the three robots. Valid for any of the robots (all use the same laser hardware and drivers).	High for our robots, low in general
Position estimator	It estimates the position of any of the robots within its environment, just using the public services of laser and motion modules (the latter for odometry).	High
Reactive navigator	It moves any of the robots to a geometric target location by using the motion and laser modules, through a reactive algorithm [32].	High

A *service* is the minimal sequential execution entity within a module. A service provides certain functionality, and can access the internal status of its module. In addition, it can request other services and thus generate communications. Services are of three types: *regular*, *event handlers*, and *notification handlers.* A regular service is requested by others or started by its own module when the module initiates (for example, a service that makes a "homing" in the industrial manipulator arm controller module). An event handler service is executed only when an asynchronous event is sent by other service (for example for detecting battery-low condition in one of the mobile robots). Finally, a notification handler service is executed only when an asynchronous notification is generated within the same module. All the services of a module can run concurrently within their module, but some of them can be set for blocking any other's execution. Table 3 shows a list of services provided by the *SENA Robot Motion* module. More details on the type, characteristics, and utility of services can be found in [7].

Regular services can have input data (parameters for the execution of the service) and output data (results of its execution). In addition to the output data of a service, the requester receives a package of information that includes communication errors, module faults, or any other anomalous situation. Also, each service can specify a priority of execution relative to the priorities of other services of its own module, which is part of the hierarchical relative priority system illustrated in Fig. 5. At run-time, as explained in Section 6, a given priority is assigned to each module (relative to the other modules' priorities). At design-time, the services within a module are divided into three priority categories: *high*, *dynamic*, and *low*. High priority services can pre-empt the execution of dynamic and low priority services of the same module. This is useful, for example, for assuring that a hardware monitoring algorithm never loses its time requirements. Low priority services can be pre-empted by any other service of the module. For example, consider a service for sending information to a web page periodically, or to update the content of a graphical application: it may be useful to assure that the service does not interrupt more critical algorithms such as hardware interaction, control loops, etc. Finally, dynamic priority services run at a priority that is a function of the priority of the caller.

Relations between structural portable entities: Two relations can be defined for connecting structural portable entities to each other: *communication*, which permits modules to transfer information between them, and *composition*, which makes up the application.

Two main types of communications can be distinguished: *signals*, for asynchronous notifications, that are implemented as requests for event handler or notification handler services, or *transmissions*, for transferring data between services synchronously, that are implemented as requests for regular services.

The composition relation is used to specify the set of modules of the application. This is fully exploited by maintenance tools, such as module repositories like the one described in Section 8.

Codification portable entities: Codification portable entities allow the programmer to *design* (not to implement) the code of services and modules. The programmer can propose codification entities in any programming language, or formalisms such as StateCharts [33]. Aracne is also intended to admit source code that can be transformed into those entities, which can help in the validation phase.

The most basic codification entity is the *code atom,* which represents a finite piece of sequential code that can be defined without using other codification portable entities. There are two types of code atoms: *portable code atoms* and *non-portable code atoms*. The former represents portable code written in some programming language. The latter represents operations that are dependent on some software or hardware. The most evident example of the latter concern communications: service requests, service responses, signalling, etc. An instance is the following

Author's personal copy

Table 3
Some of the services of the SENA robot motion module

Service	Description	Type	Input/output data
ReadADC	Read the values of the ADCs of the SENA microcontroller. At present, these ADCs are connected to the joystick, to the infrared sensors, and to the acceleration sensor.	Regular	INPUT: ⟨no data⟩ OUTPUT: vectorADC measurements
SetHeading	Indicate a heading for the robot. Non-blocking call.	Regular	INPUT: float direction_degrees OUTPUT: ⟨no data⟩
StopChair	Stops the robot.	Regular	INPUT: ⟨no data⟩ OUTPUT: ⟨no data⟩
Emergency stop	Stops the robot if a critical error occurs (i.e. a failure due to obstacle proximity).	Event Handler	INPUT: ⟨no data⟩ OUTPUT: ⟨no data⟩
ReadVelocity	Obtains the current linear velocity.	Regular	INPUT: ⟨no data⟩ OUTPUT: float veloc
ReadAngularVelocity	Obtains the current angular velocity.	Regular	INPUT: ⟨no data⟩OUTPUT: float speed
ReadPosition	Obtains the current odometric position of the robot.	Regular	INPUT: ⟨no data⟩ OUTPUT: float x, float y, float phi
ChangeVelocity	Changes the current linear velocity of the robot.	Regular	INPUT: float speed OUTPUT: ⟨no data⟩
DisableJoystick	Disables the joystick control of the robot.	Regular	INPUT: ⟨no data⟩ OUTPUT: long error
Turn	It allows turning the robot a certain number of degrees to the left (positive) or to the right (negative).	Regular	INPUT: float degrees, float turn_speed OUTPUT: ⟨no data⟩
BlockingTurn	The same as Turn, but blocking until the robot turns.	Regular	INPUT: float degrees, float turn_speed OUTPUT: ⟨no data⟩
EnableJoystick	Enables the joystick control of the robot.	Regular	INPUT: ⟨no data⟩OUTPUT: long error

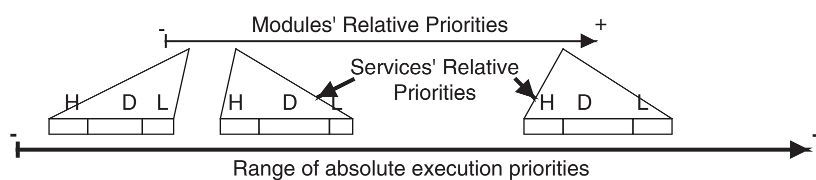

Fig. 5. The hierarchical priorities in Aracne. Each module is assigned a priority relative to other modules. Also, each service within a module is assigned a priority relative to services of the same module (H-high prioritized, D-dynamically prioritized, L-low prioritized). Final execution priorities are calculated from this hierarchical, portable scheme when the implementation of a module is produced for a given execution/real-time platform.

operation taken from the module *PERFORMER MK2 controller* of our industrial manipulator arm application, which requests a service for returning the current configuration of the arm:

NONPORTABLE_ATOM(request-synchronous&blocking, ‘‘ReadJoints’’, ‘‘PERFORMER MK2 controller’’, ‘‘ReadJoints’’, ‘‘REMOTE_PERFORMER_MK2_controller’’, ‘‘read_joints’’,’’err_var’’,’’0’’,’’INF’’).

If available, code atoms can also include time information: a time interval with the minimum and maximum expected duration of the execution of the atom (‘‘0’’ and ‘‘INF’’ in the previous example). That information can be used for validation of real-time requirements.

Relations between structural and codification portable entities: The researcher must link codification to structure in order to complete the portable design of the application. This is done through the *codification* relation.

A codification relation is defined between modules (structural entities) and codification entities. It permits us to check the validity of some important aspects of the design, for example, if time requirements in the structural part are achievable by the design of the execution logics.

This relation also permits the designer to assign more than one codification entity to a given module. In that way, Aracne captures the extendibility issue and improves reusability. In addition, fault tolerance mechanisms like

Author's personal copy

running several replicas of a module (*active replication* [34]) are enabled.

The codification relation formed in this way is a mathematical forest (a set of mathematical trees with a depth of two: their roots are modules and their leaves are codifications), called in Aracne an *Application Codification Forest (ACF)*.

4.2. The non-portable part of the design

While portable entities specify the structural design of the application, non-portable entities tie the application to its particular software and hardware dependencies. Non-portable entities in Aracne are *General Platforms* and *Particular Platforms.* General Platforms serve to classify the non-portable support necessities of the application: hardware, execution, communications, etc., and allow us to classify both commercial or non-commercial solutions already existing for robot development. Particular platforms are instances of general platforms (for example, a given operating system or a middleware solution for distributing objects). In this way, Aracne covers the wide range of possible components existing both in the literature and commercially, as was commented in the introduction. Currently, Aracne deals with five different general platforms.

A *hardware platform* (HP) represents a set of hardware devices needed for the physical execution of the application, which includes at least one processor unit and shares a motherboard.[2] This platform groups together: CPU(s), motherboard devices (hard disk, sound card, graphic card, real-time clocks, etc.), plugged-in devices (acquisition boards, automation interfaces, network interfaces, etc.), and peripherals (monitor, printer, external storage devices, etc.). For example, an application for our SENA wheelchair [6] includes two particular hardware platforms: the onboard laptop PC and the microcontroller, while an application for the SANCHO robot [32] consists of the onboard laptop and a remote computer for user interfacing and monitoring tasks, and for managing a symbolic model of the world and task planning (in SANCHO, the basic PIONEER controller platform on which the robot is mounted is considered as a monolithic, non-programmable device–that is, it is not a support for executing modules-, similarly to the arm controller of the RAM-2 robot). The industrial manipulator arm is considered to include one hardware platform: the external PC. The PERFORMER MK2 controller is considered as a hardware device that does not execute modules.

An *execution platform* (EP) represents the basic software execution environment of the application. This includes: operating systems, virtual machines, software libraries and execution libraries (e.g., interfaces with hardware devices). The particular execution platforms used in our robots include, among others, Microsoft Windows NT and XP [35] for SENA, SANCHO, and the manipulator arm; LynxOS [36], and a JAVA Virtual Machine [37] for RAM-2. The microcontroller of SENA runs no operating system or other software for supporting execution of applications.

A *communication platform* (CP) comprehends the software needed for communicating the modules of a distributed application. Its particular platforms can provide from simple protocol support (peer-to-peer, TCP/IP) to more abstract object distribution (like CORBA [22]), even a monolithic scheme (that is, no network). For example, typical SENA applications use two particular communication platforms: the ACE + TAO [38] implementation of the CORBA specification for communicating the modules that run on the onboard computer, and a USB-based communication software for connecting to the microcontroller. The PERFORMER MK2 arm uses no communication framework since the RS-232 is managed internally to the module running in the external PC (that is, it does not communicate modules).

A *real-time platform* (RTP) provides real-time facilities in software form: real-time scheduling, time measurement, sinchronization, etc. Only the facilities included in the LynxOS real-time operating system (with a POSIX 1003.1b compliant interface [39]) are needed in the RAM-2, while in SENA and SANCHO, the MS Windows OS is enhanced with the commercial RTX real-time extension by Ardence [40]. The industrial manipulator arm has no hard real-time needs in our application (low-level real time requirements are covered by its controller).

A *fault-tolerance platform* (FTP) provides software fault-tolerance facilities (in the current Aracne specification, active replication). We have currently no particular support for this general platform.

Relations between non-portable entities: A basic relation between non-portable entities, called *support*, can be defined. For instance, a hardware platform provides support for a given execution platform, which in turn provides support for a communication and a real-time platform. In that way, a set of particular platforms for a given architecture along with their support relations can be represented by an acyclic directed graph. This graph is called a *platform configuration graph* (*PCG*) in Aracne. Notice that problems like using a hardware platform that does not include a real-time clock (which would prevent any real-time platform to provide most of its supporting functionality), can be detected and corrected in the design phase, when the PCG is defined, by using simple verification tools. Fig. 6 illustrates three PCGs for applications on RAM-2, SENA, and the PERFORMER MK2 arm. Notice that the manipulator application has a simple PCG since modules only execute in the external PC. More complicated settings can be considered that are connected through (may be real-time) links to the PC and run different control algorithms, without much modification of the design of the application. Also, the generation

[2] In a microcontroller, as is the case with the SENA wheelchair, the "motherboard" typically includes just the CPU, main memory, and I/O facilities.

Author's personal copy

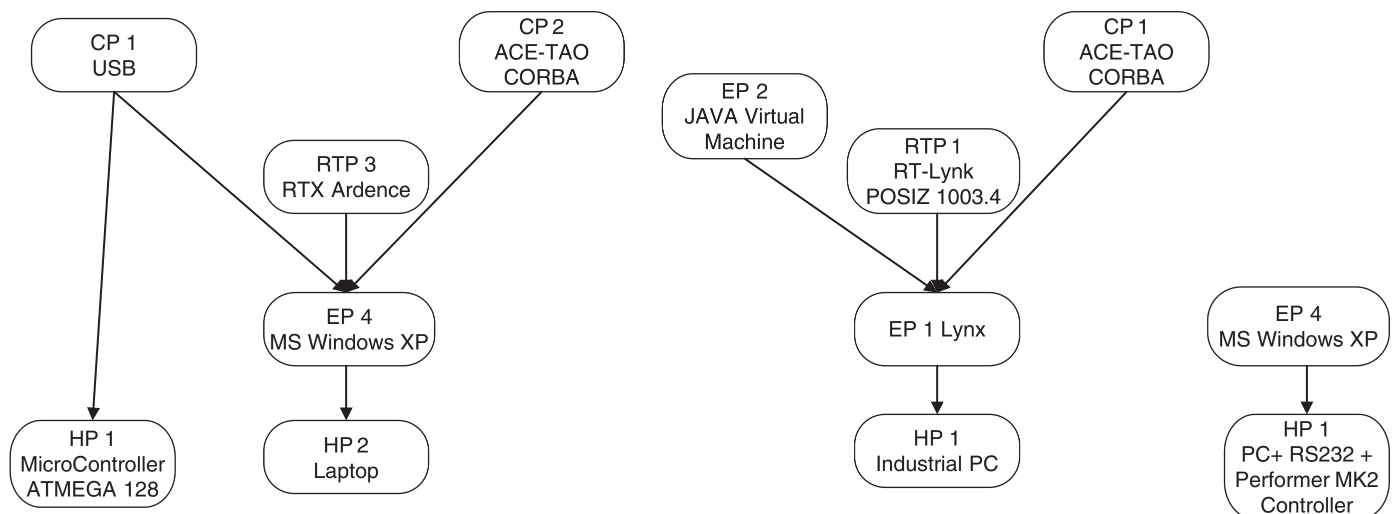

Fig. 6. PCGs for three robotic applications: the mobile robot SENA (on the left), the mobile manipulator RAM-2 (on the middle), and the PERFORMER MK2 manipulator arm (on the right). Nodes of the graphs represent particular platforms; edges represent the support relation.

of modules that run directly on the PEFORMER MK2 controller is an option (that would be analogous to the case of the module that runs in the microcontroller of the SENA robot and communicates to the laptop through an USB connection).

Any number of PCGs can be proposed for the same Aracne-based design. Each PCG allows the application to be implemented on a given set of non-portable components (robots, computers, network, software, etc.). The key point of this scheme is that portable entities are kept unchanged from one PCG to another, that is, the portable design of the architecture—modules, services, codification—is reused in the maximum number of physical robots that is possible (as has been the actual case in our mobile robots). If a PCG does not provide enough support for all the requirements, the researchers will know at design time that the corresponding implementation will exhibit reduced capabilities, or that it will be unsuitable at all.

4.3. Integrating the portable and the non-portable

Previous sections have presented most of the design aspects of an Aracne-based application. However, the design phase is not complete until portable entities are integrated with non-portable entities. Establishing those relations completes the specification and also allows the programmers to produce executable implementations.

Relating portable to non-portable entities consists basically of defining relationships between modules and the execution/hardware platforms where they are going to be executed. This is called an application configuration relation (ACR). Fig. 7 depicts the ACR of an automatic delivery service application for SANCHO. The application is an automatic indoor object delivery service, with multi-user support (notice that this is similar to an AGV, therefore with few changes we could made use of the same design for an industrial application). SANCHO must move within our building (a typical office scenario) carrying diverse objects from place to place (through reactive navigation and route planning), and returning to a recharging station whenever its batteries fall below a certain threshold. Users can request SANCHO to take an object from their places and deliver it to a given destination in two different ways: through a web interface (an applet [37]) that connects to the "Task Manager" module of the robot via TCP/IP, or through verbal commands when the robot is nearby. The delivery tasks are planned by the "Task Planner", that runs in an external computer for optimizing computational efficiency (the "MAH-Graph" module, which holds a symbolic model of the environment—topological map—for planning tasks, is also executed in that server). The planned tasks are carried out by the "Task Executor". The robot is able to speak to humans for confirming the delivery of objects or to ask for help when an unrecoverable situation occurs (for example: its location in the environment is undefined or the batteries level is critically low).

The ACR diagram enables validation tools to detect several problems before execution: no way of satisfying requirements (real-time, fault-tolerance, communications), possibility of creating reduced-performance implementations, etc. In addition, restrictions on the portability of the portable design can be stated, typically the need of some particular platform for a given codification.

Two types of relations can exist in an ACR: *portable configuration* and *non-portable configuration*. When a module is associated to a machine by *portable configuration*, we are indicating that we wish to generate an implementation for that platform, although another implementation could be possible. When a *non-portable configuration* relation is used instead, we are indicating that the module codification is tied to that platform, and thus

Author's personal copy

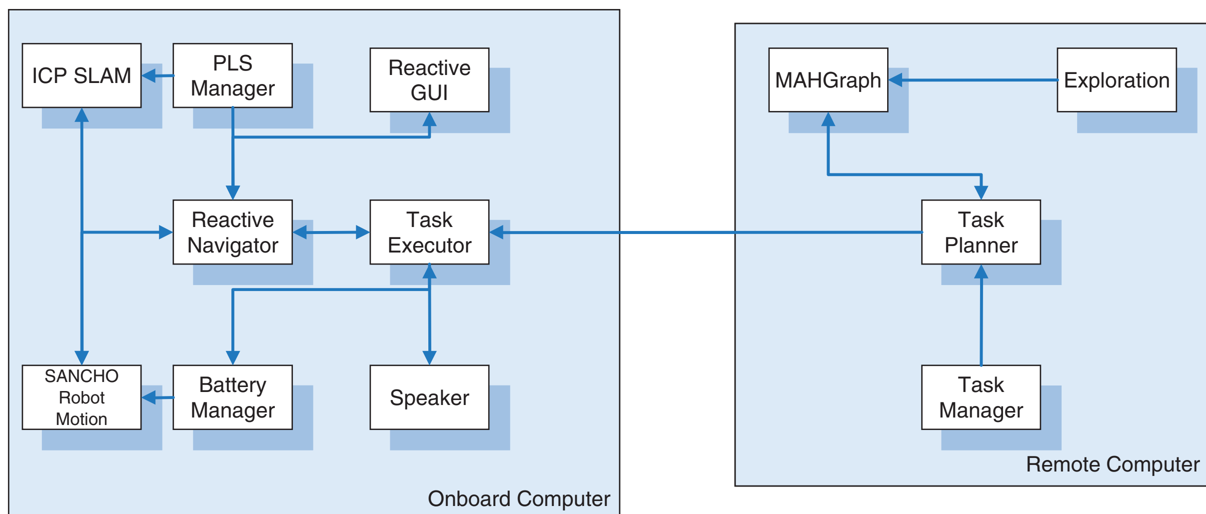

Fig. 7. ACR for the implementation of an automatic delivery application on the robot SANCHO, mixed with some communication relations between modules that clarify the diagram.

we cannot generate implementations if that platform is not present.

5. Implementation of applications: the BABEL MD

This section presents a tool that spans through the design and implementation phases of the software lifecycle: the BABEL MD. The MD allows the programmer to design modules and services using the Aracne specification explained in Section 4 and to implement them for some particular platforms. It has been previously presented elsewhere [2] in the more restricted context of our older NEXUS specification. Currently, it is enhanced for coping with Aracne, and has become the central tool of the BABEL development system.

The MD is a CASE tool that facilitates the designing of portable entities (modules and services), being also able to transform automatically a design into an implementation. The main features of the MD are:

- Visual design. It provides a user-friendly integrated development environment for designing modules, specifying their public interfaces, services, and codification (see Fig. 8). The tool satisfies the Aracne specification as presented in this paper.
- Semiautomatic generation of implementations. The programmer only needs to design a module as well as the codification of the service routines. The MD automatically generates the software for converting this specification into a complete executable program and for the integration of this program into a (possibly distributed) robotic application composed of other modules.
- Inclusion of particular platforms. The MD includes the possibility of generating implementations for a given set of particular platforms. It also allows the designer to specify which platforms must be included in the implementation due to some non-portable configuration. The MD is currently able to generate executable code for different programming languages (C, C++, and JAVA). The portable entities designed with the MD can access the facilities of two particular real-time platforms, the LynxOS real-time operating system (that is, POSIX 1003.1b compatible), and RTX for MS Windows [40]. It also supports ACE+TAO CORBA as well as a communication platform for USB. Finally, it generates executables supported by four different execution platforms: MS Windows, the JAVA Virtual Machine [37], LynxOS, and the SENA microcontroller. Notice that some of them provide hard requirements while others have relaxed dependability.

Fig. 8a illustrates the design of the structural part of a module. Fig. 8b shows an example of codification of a service. The MD starts from the source code introduced by the programmer, along with some macros that represent the most common non-portable code atoms (as depicted in Section 4.1). These macros can be entered through the use of assistants and wizards.

6. Execution of applications: the BABEL EM

This section presents a tool aimed to provide support for the execution phase of the lifecycle: the EM. In BABEL, an application is a set of modules, hence the EM serves to execute distributively and coordinately these modules (some videos of the execution of applications developed with BABEL can be found at http://www.babel.isa.uma.es/babel).

The EM is managed by a human operator from a given computer, although the tool must be running previously in every node of the network that is going to execute modules,

Author's personal copy

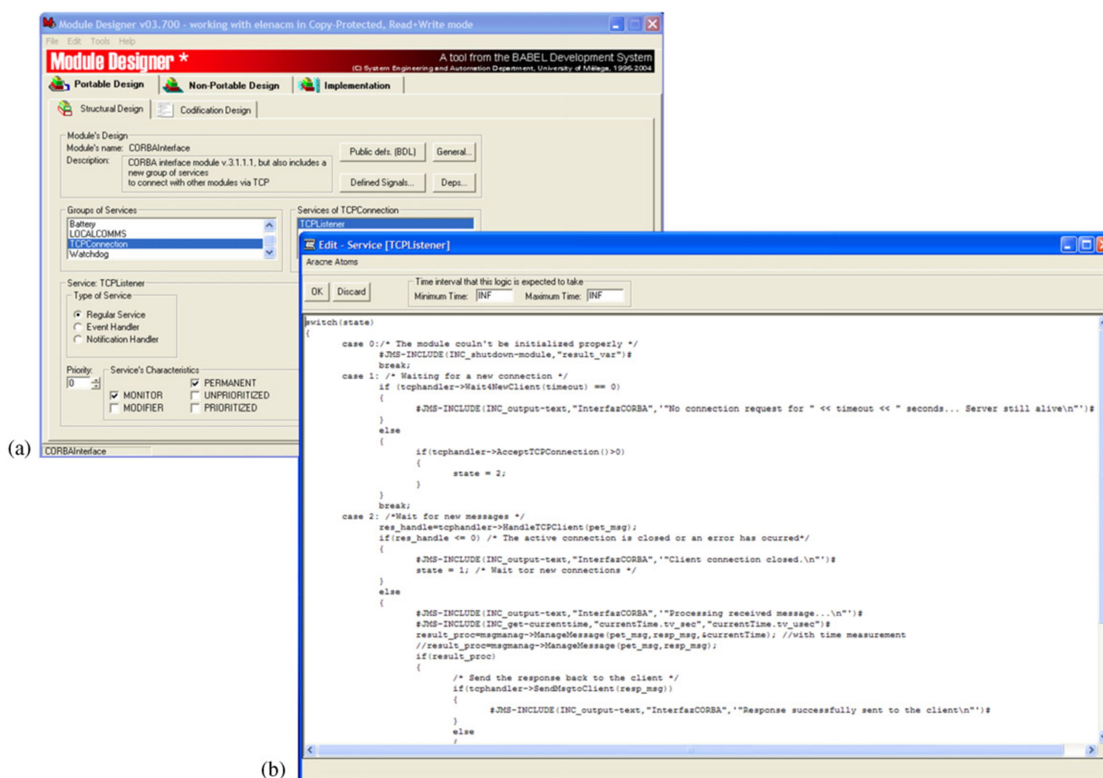

Fig. 8. Some snapshots taken during the design of a module using the BABEL MD: (a) in the background, design of the structure portable entities: (b) in the foreground, design of the codification portable entities (code of a service).

in order to coordinate the execution of the modules in all those computers. Each time a new EM is launched on a computer, it searches for the rest of the EMs in the network through multicasting [41] and connects to them through TCP permanent connections [42], hence all the EMs maintain an updated and operative image of the application network; analogously, when an EM is stopped, the rest of the network is automatically notified about that event and the images updated again.

From one of the EMs, the user can examine the modules ready for execution on any computer[3] (see Fig. 9). From these lists, he/she can select the ones included in the application and set their relative priorities and the order in which they must execute, making up a so-called "execution sequence" (see right part of Fig. 9). That sequence represents the application in the execution phase of the lifecycle, and may be saved for later use. Each module is assigned a priority with respect to other modules (for enabling real-time scheduling if some real-time platform is available).

In the execution sequence the user can also set the timing for the starting-up of the different modules: some can be set to run after certain time from the previous one (or maybe simultaneously), or can be set to execute after user prompt in the EM. Once these timings are set, the application can be launched. The EMs located in the other computers of the network receive the events registered in the sequence through their permanent connections and launch the respective modules.

Finally, the application can be stopped at any time from the same location in which it was launched. At that moment, the EM can retrieve testing information recorded by every module during execution for providing it to the BABEL Debugger (described in Section 7).

7. Debugging applications: the BABEL Debugger (D)

The BABEL Debugger is a tool that recovers information recorded during the execution phase of the lifecycle (as mentioned in Section 6) and allows the user to navigate through that information in a graphical setting. See Fig. 10 for a real example of debug information shown to the user. It works coordinately with the EM tool, which retrieves the information produced by all the modules of the application, and with the Aracne especification, which allows the programmers to set which debug information is desired to be produced. The Debugger can serve for visualizing the following information:

- Timings of the communications in the executed application.
- Sequences of execution of the different services of the modules.

[3]The EM tool also includes a basic messaging service for user coordination.

Author's personal copy

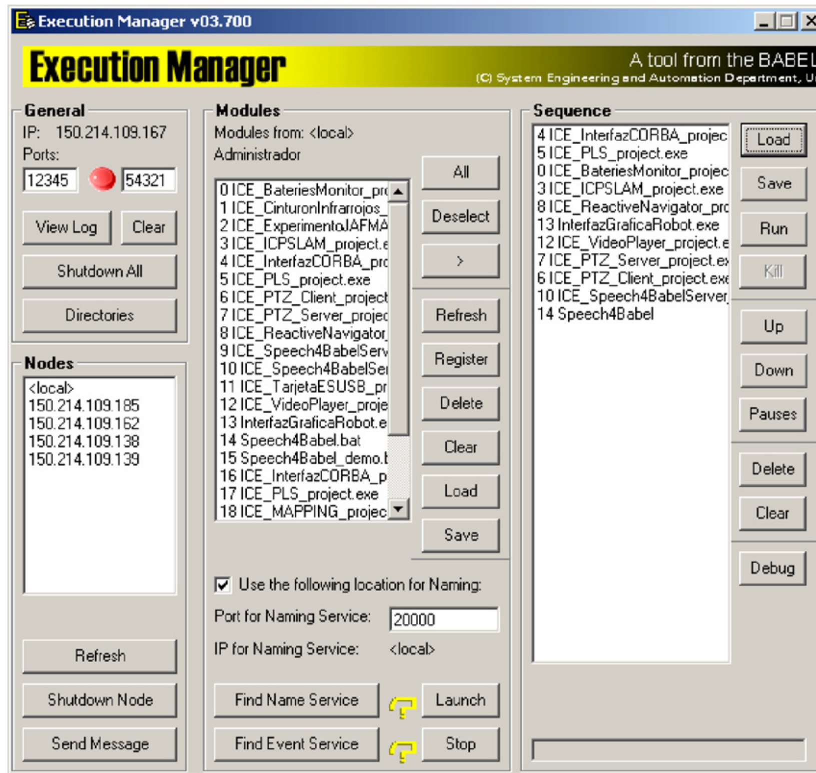

Fig. 9. Snapshot of the BABEL EM. On the left, the list of current active computers (their IPs) in the network. On the middle, it is shown the names of the modules ready for execution in a given computer (the local one in this example). On the right, a sequence of execution of modules from different machines has been entered by the user.

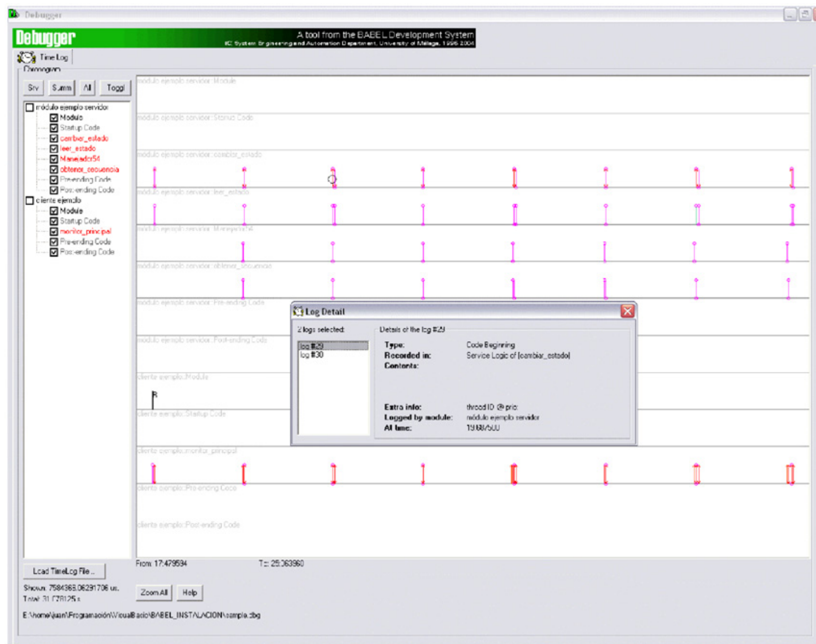

Fig. 10. The BABEL Debugger. The time logging produced by a real execution of one of our applications is shown (the logged events are drawn as lines of different colors at appropriate points in the timeline). The popup dialog offers information of some of these events.

- Programmers' messsages along with their timings. The programmers can include special non-portable atoms (see Section 4.1) in the code for tracing the execution of the code.
- Scheduling of the service taks in a given computer. It can be observed whether the modules executed on a computer can satisfy their timing requirements under the hierarchical priorities assigned by the EM.

Author's personal copy

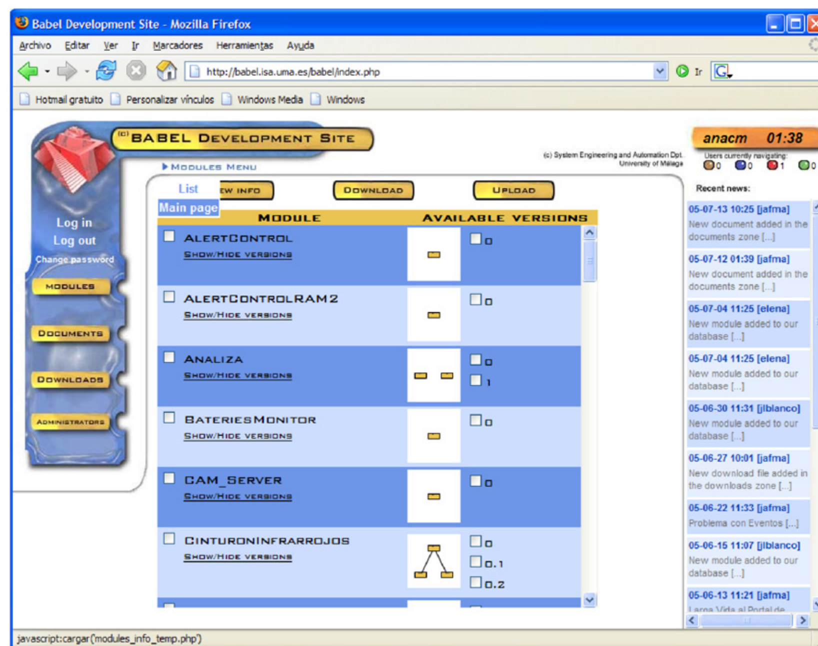

Fig. 11. The BABEL Development Site. This is a snapshot of the section dedicated to list all the maintained modules and their characteristics. Close to the modules' names are their respective version graphs (each module may have several versions).

8. Maintenance of applications: the BABEL DS

Through this section we present the most recent addition to our framework: the BABEL DS (http://www.babel.isa.uma.es/babel), a web site intended to work as a repository of code—that is, modules designed by following the Aracne specification—that also provides maintenance and some simple validation tools to keep a well-classified software and to check its correctness, respectively.

The access to the DS is restricted in order to manage our research projects coordinately, allowing the manipulation of the information related to the modules only to certain registered users. However, some sections like the listing of developed modules and the documentation are open to all visitors for evaluating the framework. In particular, the documentation section contains several videos of the applications developed up to now with the framework. Also, a version of the BABEL system that works with a set of particular platforms (those most intensively tested) is available for public download. The admittance control in the restricted areas is implemented by the utilization of two security components: groups of users, and IP address checking. The joint use of both elements warranties the safeness of the BABEL Development Site in a multi-user environment opened to the internet.

The core of the DS is a database which contains all the information about the modules. This information is available by means of a simple multi-user interface (Fig. 11). As different versions of the same module can coexist, those versions must be organized in such a way that they can be unequivocally identified, and the relationships between them are completely known. For this reason, a version control system, destined to the maintenance phase of the robotic software lifecycle, has been built.

Furthermore, two validation tools are also available in the DS. The former makes possible the detection of dependencies between modules, that is, which events and services of other modules are used by a specific module. With these data, the DS builds the so-called dependency tree, a graphical representation of the relationships of a module with the rest of the existing modules in the code repository (Fig. 12), that helps to detect anomalies by visual inspection (and also to retrieve the correct set of modules for generating a given implementation). The second validation tool is based on the first one, since it uses dependencies trees in order to reveal some other errors which cannot be detected in the design phase of the module. To be precise, this second validation tool fixes three types of errors: a module requests services or events which do not exist in the code repository, so its proper execution would be impossible; a service does not fulfill its time restrictions, and hence the desired real-time behavior of the module is not obtained; and finally, a loop is detected in the dependencies tree of a module.[4] Through both validation tools, the programmers can easily understand the relationships among the modules stored in the database, and ensure that their execution is really possible.

The BABEL DS also provides to its users with a notification and news system, as well as a DS documents and files collection available for downloading. Moreover, the DS supplies to its administrators with all the

[4]Though this situation is not strictly an error, it is considered anomalous.

Author's personal copy

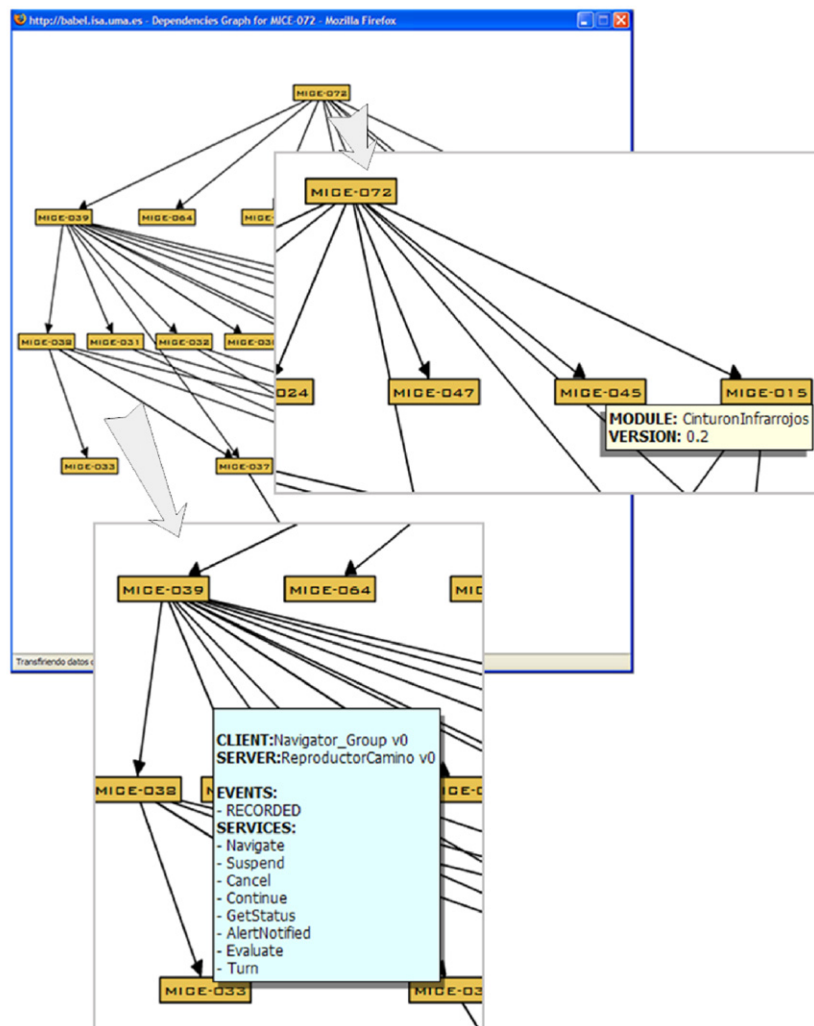

Fig. 12. A dependency tree generated by the BABEL Development Site for a module. The zoomed windows show some information that can be consulted from dependency arcs (service requests) and nodes (modules) of the tree.

mechanisms required for handling the DS itself, including statistical data about visits and utilization of the web site.

9. Conclusions and future work

In this paper we have presented a new approach to the Robotic Software Integration Problem (RSIP), called BABEL, which aims to cover all the phases of the software lifecycle under a software engineering perspective that deals appropriately with a high degree of heterogeneity. We have described the Aracne specification as the core of our approach, which establishes the design guidelines of any robotic application, covering the wide diversity of necessities that a robotic project usually involves (real-time requirements, hardware integration, software reuse, etc.), mainly through the concept of supporting platforms and the separation between the portable and non-portable parts of the application. The main idea that underlies the paper is that RSIP and its main cause, heterogeneity, are unavoidable issues in large-scale software robotic projects, and that they must coexist with stringent issues (i.e., dependability). BABEL is flexible enough for achieving the aforementioned objectives in a clear and orderly fashion, as it has been demonstrated in our research during the last years and illustrated in this paper. In fact, it has become a necessary framework for organizing the work of large-scale projects in our Department and for speeding up the development of new applications.

It is difficult to evaluate a software development framework since no standardized methodology or benchmark that captures all the relevant aspects in the software development lifecycle exists. In fact, most of the literature on software metrics is based on case studies where metrics programs are introduced in particular organizations, but rarely such analyses are compared to each other in order to come up with general conclusions [45]. Nevertheless, in this section we describe a few general and particular results that show the benefits of BABEL.

Firstly, some differences between using a modular, heterogeneity-enabling framework and those that do not

Author's personal copy

cover some of these aspects are evident. Fig. 13 depicts two graphs that illustrate these differences. As it can be seen, the development cost can be drastically reduced by using both approaches simultaneously (modular design and heterogeneity enabling).

More concretely, the main objectives of BABEL are: (i) to produce solutions that satisfy typical robotic requirements, and (ii) to reduce as much as possible the cost of development and maintainance (mostly time). For example, the modules developed for the delivery application of SANCHO (described in Section 4.3) are shown in Table 4, along with estimates of the programming time involved in each one. The table shows good development times (with some remarkable cases of a few days), particularly if we consider that they have been developed by different people (undergraduate students, doctoral students, researchers, etc.) and that these times include the design, the programming, and the testing. Also, notice that once the modules are finished, they can be reused in other robots and/or applications often at no cost (on the average, each module is reused in two of our mobile robots). For all the modules that have more than one robot in the third column we would have needed to multiply their development time if they would have been implemented from scratch for each robot. Thus, on the average, the development from scratch would have been more than twice the duration shown in the table.

Currently, we are extending the BABEL framework with the inclusion of more particular platforms (as they become available for our robotic projects), in particular, new communication and execution platforms. We are also working to include formal validation tools.

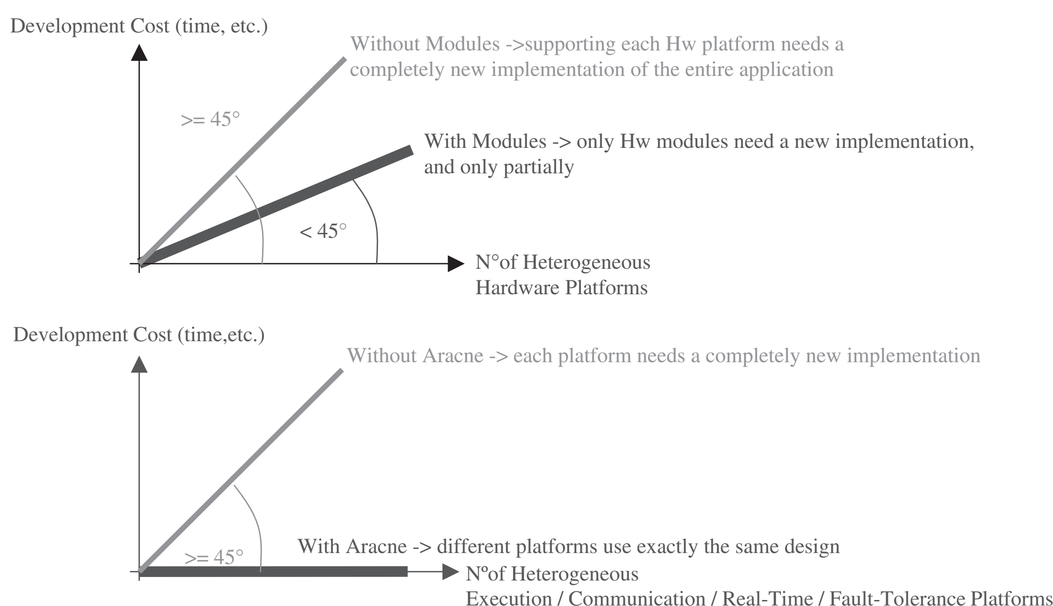

Fig. 13. Advantages of a modular and heterogeneity-enabling framework with respect to others. (Up) Effects of modularity. (Bottom) Effects of enabling heterogeneity.

Table 4
Estimates of the time of programming spent in some modules designed and implemented with BABEL for our SANCHO delivery service

Module name	Development time (Estimate)	Reused in… (ALL = SENA + SANCHO + RAM-2)
'Batteries manager'	1 day	SANCHO, SENA
'Exploration'	3 days	ALL
'GUI for reactive navigator'	3 days	ALL
'ICP SLAM'	3 days	ALL
'MAHGraph'	2 weeks	ALL
'PLS laser manager'	1 week	SANCHO, SENA
'Reactive navigator'	1 week	ALL
'SANCHO robot motion'	2 weeks	SANCHO
'Speaker'	3 days	SANCHO, SENA
'Speech Server'	1 week	SANCHO, SENA
'Task executor'	2 weeks	ALL
'Task planner'	3 weeks	ALL

Each module has been programmed by one researcher.

Author's personal copy

References

[1] Fernández-Madrigal JA, González J. NEXUS: a flexible, efficient and robust framework for integrating the software components of a robotic system. Belgium: IEEE ICRA Leuven; 1998.

[2] Fernández-Madrigal JA, González J. A visual tool for robot programming, 15th IFAC world congress on automatic control. Spain: Barcelona; 2002.

[3] Galindo C, González J, Fernández-Madrigal JA. A control architecture for human-robot integration. Application to a robotic wheelchair. IEEE Trans Systems, Man Cybernet—Part B 2006; 36(5), [in press].

[4] Fernández-Madrigal JA, González J. The NEXUS open system for integrating robotic software. Robotics and computer-integrated manufacturing 1999;15(6).

[5] Galindo C, Fernández-Madrigal JA, González J. Improving efficiency in mobile robot task planning through world abstraction. IEEE Trans Robot 2004;20(4):677–90.

[6] Fernández-Madrigal JA, Galindo C, González J. Assistive navigation of a robotic wheelchair using a multihierarchical model of the environment. Integr Comput-Aided Eng 2004;11(4):309–22.

[7] Fernández-Madrigal JA. The BABEL development system for integrating heterogeneous robotic software. Tech. rep. System Engineering and Automation Department, University of Málaga. 2003.

[8] Larman C. Applying UML and patterns: an introduction to object-oriented analysis and design and the Unified Process, second ed. Upper Saddle River, NJ: Prentice-Hall; 2002.

[9] Coste-Manière E, Turro N. The MAESTRO language and its environment: specification, validation and control of robotic missions. Grenoble, France: IEEE/RSJ; 1997.

[10] Alami R, Chatila R, Espiau B. Designing an intelligent control architecture for autonomous robots. Tokyo, Japan: International conference on advanced robotics; 1993.

[11] Simmons R, Lin L-J, Fedor C. Autonomous task control for mobile robots (TCA). Fifth IEEE international symposium on intelligent control, Philadelphia, 1990.

[12] Albus JS, Quintero R. Towards a reference model architecture for real-time intelligent control systems. IEEE international symposium on robotics and manufacturing, Burnaby, Canada,1990.

[13] Cimetrix Incorporated (2005). The CODE programming interface. http://www.cimetrix.com

[14] Real-Time Innovations Inc. (RTI) & Stanford University Control shell: object-oriented framework for real-time system software, 2005. http://www.rti.com/

[15] Fleury S, Herrb M, Chatila R. GeNoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. Grenoble, France: IEEE/RSJ; 1997.

[16] Gentleman WM, MacKay SA, Stewart D, Wein D. An introduction to the harmony realtime operating system. Newsletter of the IEEE computer society technical committee on operating systems.

[17] Stewart DB, Khosla PK. The chimera methodology: designing dynamically reconfigurable and reusable real-time software using port-based objects. Int J Software Eng Knowledge Eng 1996;6(2):249–77.

[18] Pack RT, Wilkes DM, Kawamura K. A software architecture for integrated service robot development. IEEE conference on systems, man, and cybernetics, Orlando FL, 1997.

[19] Brugali D, Fayad ME. Distributed computing in robotics and automation. IEEE Trans. Robot Autom 2002;18(4).

[20] Flanders Mechatronics Technology Centre The OROCOS Project, 2005.http://www.orocos.org/

[21] Pritschow G, Altintas Y, Jovane F, Koren Y, Mitsuishi M, Takata S, Van Brussel H, Weck M, Yamazaki K. Open controller architecture—past, present and future, Ann Cirp 2001, 50/2/2001.

[22] Henning M, Vinoski S. (1999). Advanced CORBA Programming with C++, Addison-Wesley Professional, ISBN 0201379279.

[23] Sunrise medical 2005. http://www.sunrisemedical.com

[24] Gonzalez J, Ollero A, Reina A. Map building for a mobile robot equipped with a laser range scanner. San Diego, CA, USA: IEEE ICRA; 1994.

[25] Reina A, Gonzalez J. A two-stage mobile robot localization method by overlapping segment-based maps. Robotics and autonomous systems, Vol. 31. Amsterdam: Elsevier Science; 2000.

[26] Muñoz AJ, Gonzalez J. 2D landmark-based position estimation from a single image. Leuven, Belgium: IEEE ICRA; 1998.

[27] Text to speech software. Second Speech Center, 2005. http://www.zero2000.com

[28] IBM viavoice. 2005. http://www-3.ibm.com/software/voice/viavoice/

[29] Active media. 2005. http://www.activrobots.com/ROBOTS/index.html

[30] precision microcontrol corporation 2005. http://www.pmccorp.com/

[31] Eshed Robotec LTD. 2005. http://www.intelitek.com/

[32] Blanco JL, González J, Fernández-Madrigal JA. The trajectory parameter space (TP-Space): A new space representation for non-holonomic mobile robot reactive navigation. Beijing (China): International Conference on Intelligent Robotic Systems (IROS); 2006.

[33] Harel D. StateCharts. A Visual Formalism for complex Systems. In Science of Computer Programming 1987;8:1–4.

[34] Guerraoui R, Schiper A. Software-Based Replication for Fault Tolerance. IEEE Computer. 1997;30(4).

[35] Microsoft (2005). http://www.microsoft.com/

[36] Lynx Real-Time. (1993). LynxOS Application Writer's Guide; Lynx RTS.

[37] Sun's JAVA homepage (2005). http://www.sun.com/java/

[38] Schmidt D. ACE+TAO Corba Homepage, 2005. http://www.cs.wustl.edu/~idt/TAO.html.

[39] Gallmeister B. POSIX 4. Programming for the real world, O'Reilly & Associates, ISBN 1-56592-074-0, USA. 1995.

[40] Ardence. 2005. http://www.ardence.com

[41] Sanjoy P. Multicasting on the internet and its applications. Dordrecht: Kluwer Academic Publishers; 1998.

[42] Taylor E. TCP/IP complete. McGraw-Hill Professional; 1998.

[43] Collett THJ, MacDonald BA, Gerkey BP. Player 2.0: toward a practical robot programming framework. In: Proceedings of the Australasian conference on robotics and automation (ACRA'05), Sydney, Australia; 2005.

[44] Intelitek. 2006. http://www.intelitek.com/

[45] Gopal A, Krishnan MS, Mukhopadhyay T, Goldenson DR. Measurement programs in software development: determinants of success. IEEE Trans Software Eng 2002;28(9).

[46] JAI Mechademic Company. 2006. http://www.jai.com/

[47] Matrox Imaging Company. 2006. http://www.matrox.com/imaging/