

A Suite of Petri net based Tools for Monitoring and Debugging Distributed Autonomous Systems

Joaquín López, Diego Pérez, Alejandro Santana-Alonso and Enrique Paz

Dep. Ingeniería de Sistemas y Automática,
University of Vigo, Campus Universitario.
C/Maxwell s/n 36200 Vigo, Spain
{joaquin, dplosada, asantana, epaz} @uvigo.es

ABSTRACT

This paper describes the use of Petri nets in a suit of tools to design, analyze, monitor, log and debug the run-time coordination of distributed control systems. The interaction between the distributed components is modeled using Petri nets. Before running the application, different Petri net analysis tools can be used to analyze the system. While running, interactions between the different components of the distributed system can be easily monitored watching the evolution of the different Petri nets. Besides monitoring, the system can be instrumented to log state changes with different levels of detail. Finally, graphical tools can be used for visualizing the evolution of the system step by step or at the same running pace. These tools allow also go to a specific point of the execution and visualize the state of the global system.

Keywords: run-time verification, coordinating distributed systems, Petri nets, debugging distributed systems.

1. INTRODUCTION

A distributed application is a set of processes executed on different machines across a network. The improvement in communication infrastructures has contributed to the development of more and more distributed systems. Debugging and verifying distributed control programs is notoriously difficult because of their distributed nature. Obtaining a snapshot of a large scale distributed system on an asynchronous communication infrastructure for complex applications might be very costly. Still, distributed control programs are becoming more and more common for complex applications that include process control [1], robot applications [2] and production plant control [3], among others.

There are various approaches to debugging distributed systems. They include: model checking, collecting and analyzing logs, replay debugging, runtime verification and using virtual machines. Using model checking, a model of the program is created and then explored to verify the design specification of the distributed system and the implementation. Examples of model checking are MaceMC [4] and CrystalBall [5]. Collecting logs from different processes and analyzing them off-line to compare with the expected behavior is used in Pip [6] and [7]. Replay debugging tools log the execution of the application nodes and replay them deterministically. In order to be able to replay the execution some tools, such as liblog [8], need to log every message and application state. Runtime

verification is a technique that combines formal verification and program execution. Debugging with virtual machines uses a debugger placed in a virtualization layer above the hardware but beneath the operating system.

A conventional approach to distributed programming [9] involves sequential processes that send and receive messages. In distributed control, these messages are mainly related to commands and events that coordinate their execution in order to carry out a defined task. In some applications such as those used in many production plant controls, one of the modules is in charge of the global coordination. For these cases, even though each process works autonomously, they will receive events and commands from the coordination module. Each individual process may be analyzed independently. Most of the existing debugging tools can be used to debug local errors. Our approach provides Petri net based tools for designing, analyzing, logging and debugging the coordination between the different processes. This in turn determines which process is not working as expected. We propose the use of existing tools together with a coordination replay simulator to debug each individual process.

The rest of the paper is organized as follows. The next section overviews the general structure of the distributed control system. Section 3 describes the coordination between the different control units. Section 4 presents the model checking as a first way of debugging. Section 5 explains the second way of debugging that consists on monitoring the system during execution. Section 6 describes the off-line debugging analyzing the logged data. And finally, section 7 ends the paper with the results and conclusions.

2. OVERVIEW

In a distributed control system, the controller elements are not central in location. Rather, they are distributed throughout the system with each component sub-system controlled by one or more control units. The entire system of controllers is connected by networks for communication and monitoring.

In the example on figure 1, an industrial robot picks up boxes from a conveyor belt and leaves them on top of a mobile robot. Then, the mobile robot delivers the boxes throughout the building. There is a computer that controls the conveyor belt and sends signals to the industrial robot (pick up control unit).

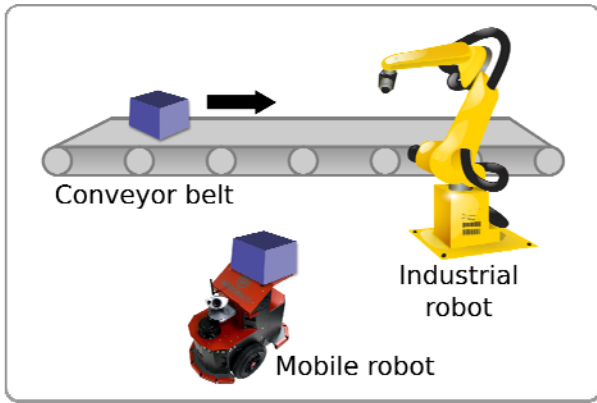


Figure 1. Pick up cell. An industrial arm gets boxes from a conveyor belt and places it on top of a mobile robot.

The mobile robot has its own onboard control unit. There is a third controller computer for the Building Automation System (BAS) that is in charge of opening and closing doors and controlling the elevators. The mobile robot needs to communicate with the BAS to request that doors be opened and closed and to manage the elevators. The pick up control unit needs to coordinate with the robot in order to wait for the mobile robot to be ready to leave the boxes.

Figure 2 shows the distributed control units and the coordination and debugging system presented in this paper. There are four different control units that autonomously control different elements of the system: the robot, the building devices, the user interface and the pick up cell (conveyor belt and industrial robot). The central server is in charge of the coordination of the distributed control units for the execution of different tasks. There are three modules running on the central server:

- **Petri net editor.** This module allows the definition of tasks as Petri nets. These tasks include commands that might be executed by the four control units. We use hierarchical Petri nets where a command in a Petri net can be the execution of another one. Events can be produced by any control unit and any other task (Petri net). Finally, Petri nets can be analyzed for properties such as safeness, boundedness and liveness.
- **Petri net dispatcher.** The tasks are executed by the dispatcher that loads the Petri nets from the XML definition file. Then interprets the Petri net executing the commands according to the current marking and subscribes to events that might evolve the loaded Petri nets. The debugger can also log different information such as the evolution of the Petri nets, the commands sent and the events subscribed to at different levels of detail.
- **Petri net debugger.** While executing tasks, the evolution of the associated Petri nets can be monitored using this debugger. Besides, the debugger is an almost necessary tool for analyzing the logs and visualizing the system evolution at the execution running pace or jumping to a specific point.

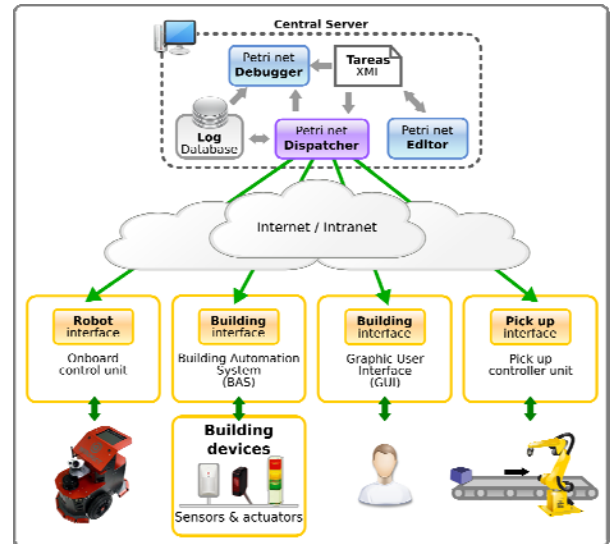


Figure 2. Distributed system architecture. Four distributed control units are coordinated by the central server.

3. COORDINATING THE CONTROL UNITS OF THE DISTRIBUTED SYSTEM

Petri nets have been widely used to model, design, execute and evaluate tasks in manufacturing dynamic systems. In this work we use hierarchical binary interpreted Petri nets.

As a simple example, figure 3 shows the Petri net that can be used for the “GET NEW BOX” task. There is only one initial mark in the place labeled “INIT”, while the “END” place has been selected as a final place. The task ends when only the final places are marked. It can also end when there are no marks on the Petri net if no final places have been defined.

First transition (“Connected”) is fired when both control units are connected. In this case two places become marked: the “Arm gets box” commands the pick up control unit to get the next box and the “Robot pick up position” commands the mobile robot to move towards the pick up position.

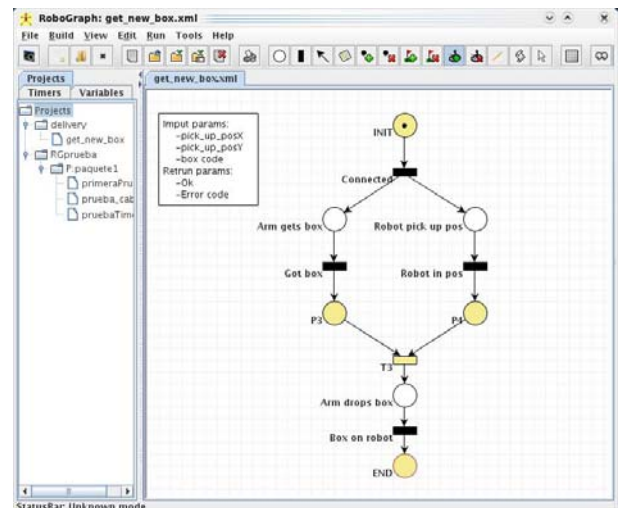


Figure 3. RoboGraph GUI editing a Petri net that coordinates the mobile robot control unit and the pick up control unit to execute the GET NEW BOX task.

Both actions will be executed at the same time. When the arm gets the box and is ready to release it, the corresponding event will be published and the “Got box” transition will be fired. On the other sequence, when the robot reaches the pick up position, it will publish the position reached event and the “Robot in position” transition will be fired. Transition T4 has no event associated. Therefore, when both incoming places are marked, meaning that the arm has the box and the mobile robot is in position, T4 will be fired and the arm will proceed to drop the box on top of the robot according to the command associated to “Arm drops box” place. When the box is on top of the mobile robot, the transition “Box on robot” is fired. Firing this transition will remove the mark from “Arm drops box” place and sets a mark on “END”. The “END” place is the only final place; therefore, the task will finish publishing the corresponding “end_task” message. Most of the commands associated to most of the places in figure 3 are messages that need to be sent to control units. However, the “robot pick up position” place executes another Petri net that needs to coordinate the robot and the building control unit to navigate the robot from the current position to the pick up position. Even though they are not included in figure 3, different Petri net mechanisms such as timers can be added to deal with some common problems.

In editor mode, the user can create new tasks using a simple and intuitive Petri net graphical editor. Figure 3 shows the GUI while editing a task. The Petri net structure is created by selecting and dragging the different elements: places, transitions, arcs and marks. Then the actions (associated to places and transitions) and conditions (associated to transitions) must be defined.

Actions can be commands implemented in any control unit in the distributed control architecture of figure 2. These commands can be selected from a menu list automatically generated by the GUI. Each command is a message and the user must define the command parameters that will automatically appear in a new dialog window when that command is selected in the editor.

When dispatcher executes the Petri net, the messages assigned to places and transitions will be published as the net evolves. Some special commands are also available, such as start and stop another task (Petri net) or start a timer.

Conditions can be events produced by any module in figure 2. These events are also selected from the menu list generated automatically by the GUI. An event can be the simple arrival of a message, a condition on some message parameter or any logical expression on several parameters over the same or different messages. RoboGraph GUI allows any logical expression to be defined over the message fields. However, complex conditions over message fields are sometimes more naturally expressed using other programming languages. For these cases, a Java-like editor is also integrated in the Petri net editor to program conditions and actions associated to places and transitions.

Timers are a tool widely used in automation that comes in very handy here. In addition, in our applications, we have also used them as an error detection mechanism in order to time some actions of different modules. Actions can start a timer while conditions can test the value of a timer.

Global variables are used to get starting data and store information to share conditions and events in different places and/or transitions.

4. ANALYZING THE COORDINATION MODEL

Petri nets have an origin dating back to 1962, when Carl Adam Petri wrote his PhD on the subject. Since that time, Petri nets have been accepted as a powerful formal specification tool for a variety of systems, including concurrent, distributed, asynchronous, parallel, deterministic and non-deterministic.

The ability to analyze Petri nets is generally considered to be the most important activity. Through the analysis of a Petri net, the designer can gain insight into the behavior and properties of the modeled system. There are two major types of analysis that may be performed on Petri nets. The first involves the creation of a reachability tree, and second involves matrix equations. There are tradeoffs involved with selecting a particular analysis technique, but the reachability tree method was first chosen for this project because it can effectively solve the safeness and boundedness properties.

A Petri net is declared safe if all of the places in the net are safe. A place is said to be safe if, for all possible markings, the number of tokens in that place never exceeds one.

A Petri net is k-bounded if, for all possible markings, the number of tokens in any individual place in the net never exceeds k. Since we use binary Petri nets, we want them to be 1-bounded which is equivalent to safe.

One interesting property regarding control systems is liveness, which is related to the complete absence of deadlocks. Liveness indicates the capability of transitions to be fired for all the possible reachable states. The concept of liveness is closely related to the absence of deadlocks. This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. However, liveness is a more restrictive condition than deadlock-free.

There are a significant number of Petri net tools that do provide analysis of the reachability tree, some of which address the properties of liveness and boundedness. Roméo [10] and CPN-AMI [11] are a couple of Petri net design and analysis applications of the many that can be found in tool databases such as [12].

We are more interested in behavioral than structural analysis because, even though it depends on the initial marking, the structural analysis is too restrictive for this application. The analysis modules implemented include:

- **State space analysis:** The Petri net is analyzed by building a tree of all the reachable markings. The reachability tree can then be analyzed to determine properties of the Petri net such as reachability, boundedness, deadlock, and safeness.
- **Invariant Analysis:** The Petri net is analyzed to determine both, the P-Invariant and T-Invariant vectors.

The structure of the Petri nets designed with RoboGraph can be stored in a Petri Net Markup Language (PNML) so that they can be analyzed by a wide set of available tools that handle this format. PNML is a proposal of an XML-based interchange format for Petri nets developed by the Petri net community as a mean to exchange models unambiguously.

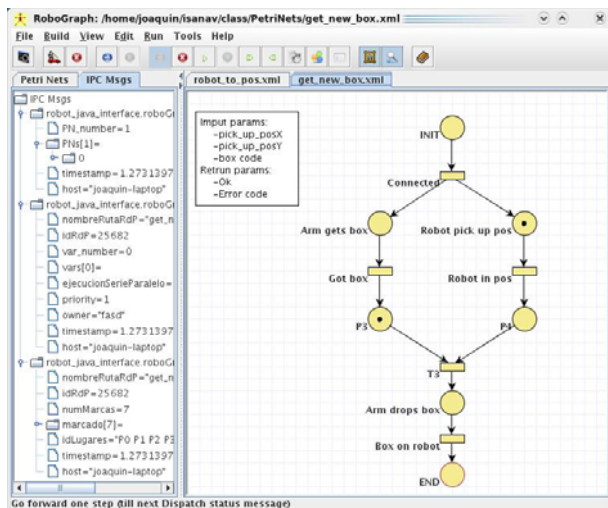


Figure 4. Petri net debugger main window.

5. MONITORING

Even if a program is based on a fault-free algorithm, its implementation can contain errors arising, among other possible sources, from coding errors. Similar situations arise in distributed system coordination. This is, we might use safe and live Petri nets but still debugging tools are needed.

Dispatcher schedules the different control unit actions and executive actions (other Petri nets), as well as the synchronization with the events produced. The interaction with other control units in the architecture is performed by publishing and subscribing to messages. This way, local problems in a control unit, such as a deadlock problem, do not block dispatcher. Also, simple mechanisms to detect and recover from a failure or exception situation can be established. When starting, dispatcher subscribes to the task execution or cancellation requests. Execution requests that cannot be executed at the reception time are stored in different queues according to their priority. A task execution request can come from different modules (figure 2), such as user interface modules (user requests) and dispatcher itself (command associated to other Petri net).

The execution of a new task starts loading the interpreted Petri net from a PNML file. Then dispatcher subscribes to all the messages referenced in the events. Finally the initial marking is set and the actions associated to the marked places are executed. The Petri net can only evolve with the arrival of messages or the end of a timer.

Each time a change in the status of a Petri net (start, stop, evolve) or in the waiting queues (new requests added or removed) is produced, a new message reporting that change is issued for monitoring (debugger module in figure 2) and stored in the log file for off-line debugging.

Online monitoring

Debugger module in monitor mode subscribes to different dispatcher messages that show the status of the different running or waiting Petri nets. Every running Petri net is shown in a different tab with the current marking as in figure 4. When dispatcher evolves a Petri net marking, a new message is issued and the debugger will update the monitor tabs. Therefore, using the monitor we can see the status of the system in a snapshot, since the marking of the running Petri nets represents its status. This is a very helpful tool when debugging an application. An

information window with the queued tasks (Petri nets) and messages can also be displayed on the left tabbed pane shown in figure 4.

6. LOGGING AND DEBUGGING

A general approach to debugging a distributed system is to create logs while the system is running and analyze them off-line to reduce interference with the execution. In the coordination system proposed here, the evolution of the coordination Petri nets, together with the messages received and published to the different control units, can be logged at different levels of detail:

- **Level 1.** Logging the status (marking) of a Petri net each time a transition is fired.
- **Level 2.** Logging data on level 1 and messages exchanged with the control units without the parameters.
- **Level 3.** Logging data on level 2 and messages including all the parameters.

When the amount of logging data is significant, then the debugging takes place in an offline manner. The search for a bug must employ effective strategies for choosing what is relevant and what is not.

The system administrator can then run the debugger in the play-logger mode, open the log file and play it at the same pace as in the real execution. Different tabs with the running Petri nets will be shown as in monitor mode. Besides the regular play option, the user can monitor the log file step by step. It can also jump to a defined place in "execution" as do many commercial programming development environments (C, Java, C++, etc.). Finally, the user can see different details about the IPC (Inter Process Communication) [13] messages including the information contained in the fields of the message.

Figure 4 shows the main debugger while replaying a logged execution. The panel on the left-hand side of figure 4 includes a couple of tabs: the Petri nets info in one tab and the messages info in another tab. The panel on the right-hand side shows two tabs corresponding to the two Petri nets that are active at that execution point shown in the figure.

Debugging the system directly from the logging data can be quite difficult. However, using the Petri net debugger is a very easy way to find the execution point where the evolution of the system does not correspond with the one for which it was designed. Besides, a great number of bugs end up with the Petri nets stuck with a fixed marking. It is quite easy to find this point and from it debug the application.

For example, when executing the Petri net of figure 3, the Petri net debugger can freeze with the marking of figure 4. With a simple look at the Petri net marking we can see that the problem is that the "GET NEW BOX" Petri net is waiting for the "mobile robot in pick up position" event. And this should be issued by the "ROBOT TO POS" Petri net. The next step is to check why the "ROBOT TO POS" Petri net is not reporting the event and we can find one of the following: the building control unit is not working properly, the mobile robot control unit is not working properly or there is some design problem in the "ROBOT TO POS" Petri net.



Figure 5. Surveillance application robot (WatchBot).

7. RESULTS AND CONCLUSIONS

The main contribution of this paper is to show the debugging advantages of using Petri nets in the design and implementation of the distributed system coordination.

First, they can be used for model checking to gain insight into the behavior and properties of the modeled system. Research on Petri net analysis has produced a number of algorithms that can be used.

Second, monitoring the system using Petri nets is very helpful, since we can observe the status of the system in a snapshot because the marking of the running Petri nets represents its status.

The third possibility is to analyze the data logged during execution. A visual interface is used to present the evolution of the Petri nets. It is also possible to jump to a specific point in the execution and see the data of each message, if necessary.

The fourth possibility is to use the log to replay the coordination system while we debug one of the units without the need to run all the others. This can be done only in the case that all the messages exchanged between the control units go through the coordination unit.

All these possibilities have been implemented in the Robotics Integrated Development Environment (RIDE) [14]. Developers use the same task programming IDE (RoboGraph) on two different levels. The first is to program tasks that must be executed autonomously by one robot and the second is to program tasks that can include several robots and building elements. This second level includes different distributed devices that might include autonomous robots and building devices such as elevators, automatic doors and sensors.

The system has been tested in a security and surveillance application recently finished and presented [15], even though a former version was already presented in [16]. The building surveillance application is a Multirobot system monitored through Internet. Each robot (figure 5) can handle some daily surveillance routine tasks. Sensor information, such as real-time images captured by a camera on the robot with pan/tilt/zoom functions, can be transmitted back to the central management office via a local area network.

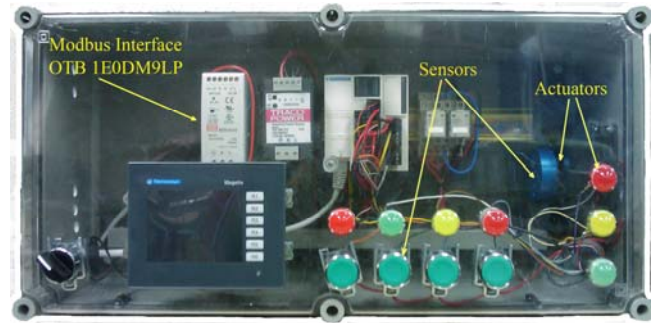


Figure 6. One Modbus module used to test the transactions between building devices and the central control system.

In this distributed application three different type of control units have been used:

- The mobile robot control unit manages mobile robot navigation and surveillance autonomous tasks. It is connected to the intranet via Wi-Fi.
- The building control unit manages the building devices including security sensors and alarms. We developed a scale model for the building sensors and actuators network. For the model, a few Modbus modules (Schneider OTB 1E0DM9LP) with some leds and buttons, connected to each Modbus interface, as shown in figure 6, were installed.
- The user interfaces manage the interactions with users allowing them to monitor the mobile robots activities and command new surveillance tasks. The main window of a user GUI is shown in figure 7.

Another application developed using RIDE was a tour guide robot that participated in the “Xuventude Galicia Net” public event for the 2007 and 2008 editions with different robots. In the 2007 edition, a Peoplebot base worked for three days in the “Palacio de Congresos y Exposiciones de Galicia”, Santiago de Compostela (Spain). In the 2008 edition, a B21 base with a robotic head (figure 8) was used. Even though only one robot was used in both cases, they were connected via WiFi to a central server as in figure 2.

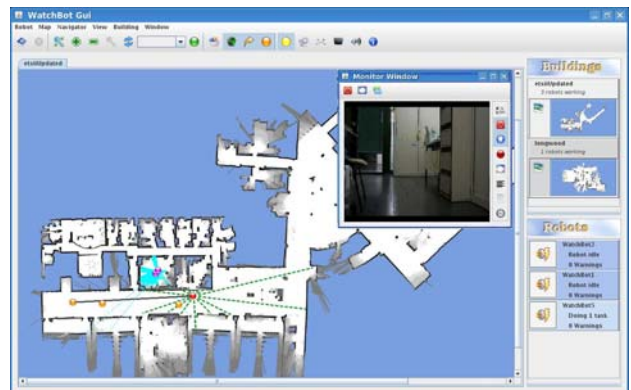


Figure 7. Surveillance users GUI

For the robot tour guide application, users can access the system from any terminal connected to the web using a GUI. In that GUI, users can see a map with the position of the robot connected to the central unit. Different icons in the toolbar will show the relevant points where the robot has to play several multimedia files. Clicking on the robot will show its status, running and waiting tasks and a little window with the camera. Finally, the menu includes options to tele-operate the robot, camera pan-tilt and other camera functions including the zoom.

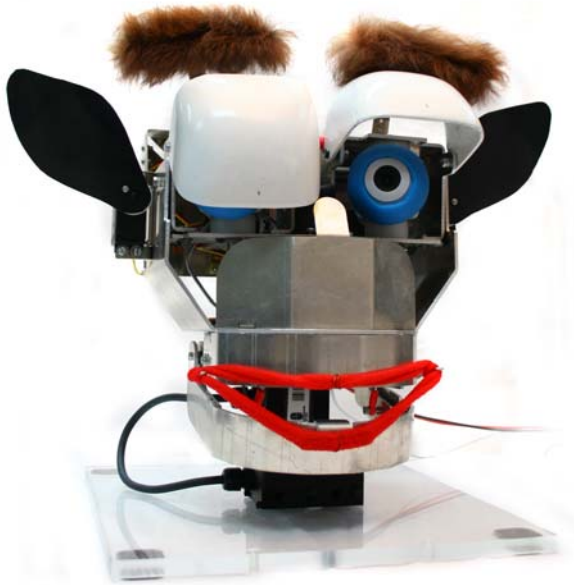


Figure 8. Mechatronic robot head for the tour guide application.

Videos showing the applications together with examples using RIDE tools can be seen at [17].

8. REFERENCES

[1] Bonasso, R.P., Kortenkamp, D., and Thronesbery, C.: Intelligent control of a water recovery system. **AI Magazine**, Vol. 24, No. 1, Spring 2003.

[2] Simmons, R., Singh, S., Hershberger, D., Ramos, J. and Smith, T.: First results in the coordination of heterogeneous robots for large-scale assembly. Proc. of **International Symposium on Experimental Robotics**, Honolulu Hawaii, Dec. 2000.

[3] Musliner, D.J. and Krebsbach, K.D.: Applying a procedural and reactive approach to abnormal situations in refinery control. in Proc. of **Conference on Foundations of Computer-Aided Process Operations (FOCAPO)**, Snowbird, Utah, July 1998.

[4] Killian, C., Anderson, J. W., Jhala, R. and Vahdat, A.: Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In 4th **USENIX Symposium on Networked Systems Design and Implementation**, pages 243-256, April 2007.

[5] Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak V.: CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. **Technical report, EPFL**, 2008. LARA-REPORT-2008-06.

[6] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat A.: Pip: Detecting the Unexpected in Distributed Systems. In 3rd **USENIX Symposium on Networked Systems Design and Implementation**, pages 115-128, 2006.

[7] Kortenkamp, D., Simmons, R., Milam T., and Fernández J.: A Suite of Tools for Debugging Distributed Autonomous Systems. **Formal Methods in System Design**, Volume 24 , Issue 2 (March 2004), pp:157-188.

[8] Geels, D., Altekar, G., Shenker, S. and Stoica I.: Replay Debugging for Distributed Applications. In **USENIX Annual Technical Conference**, pages 289-300, May 2006.

[9] Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. **PhD thesis, Royal Institute of Technology, Stockholm**, Sweden, November 27 2003.

[10] Gardey, G., Lime, D., Magnin, M. and Roux, O.: Roméo: A tool for analyzing time Petri nets. In **17th International Conference on Computer Aided Verification (CAV'05)**, Lecture Notes in Computer Science, Edinburgh, Scotland, UK, July 2005.

[11] Kordon, F. and Paviot-Adet, E.: Using CPN-AMI to validate a safe channel protocol. in the proceedings of the **International Conference on Theory and Applications of Petri Nets - Tool presentation part**, Williamsburg, USA, June 21-25, 1999.

[12] Petri net world web page. **Petri Nets Tools Database**. (<http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html>).

[13] R.Simmons, The interprocess communications system (IPC). <http://www.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html>. Accessed: 8 february 2012.

[14] Lopez, J., Pérez, D., Zalama, E.: A framework for building mobile single and multi-robot applications. **Robotics and Autonomous Systems** 59(3-4), 151 – 162 (2011). DOI 10.1016/j.robot.2011.01.004.

[15] J. L. Fernández, D. P. Losada, R. Sanz, E. Paz, Sistema de vigilancia de edificios basado en robots móviles, Workshop Robot'2009, Barcelona, Spain 2009.

[16] J. L. Fernández, D. P. Losada, R. Sanz. Enhancing Building Security Systems with Autonomous Robots. Proceedings of The 2008 **IEEE International Conference on Technologies for Practical Robot Applications**. November 9-10, 2008, Boston, USA, pp:19-25.

[17] Robotics Integrated Development Environment (RIDE). <http://webs.uvigo.es/vigobot>.