MULTI-AGENT DEBUGGING AND MONITORING FRAMEWORK

João Figueiredo, **Nuno Lau**, **Artur Pereira**

IEETA/DETI, Universidade de Aveiro
joao.figueiredo@ieeta.pt, lau@det.ua.pt, artur@det.ua.pt

Abstract: In this paper we present a framework developed for the CAMBADA Middle-sized league robotic team, which allows human developers to better understand the robots actions during a game. Robotic soccer teams are in their nature dynamic multi-process and multi-agent systems, and knowing what is happening in all processes running on the agents at the same time is a hard task. To accomplish this task we developed a framework to create log files, one per process, and to interlace them later. The logs represent robot's knowledge. The framework allows the synchronization and visualization of logs and videos. Videos give the actual real behaviors. This will allow us to understand the robot's reasoning. A GUI utility to navigate and search inside log files was also developed.

Keywords: monitoring, debug, multi-agent, multi-systems, robotics, MSL, Middle-Sized League

1. INTRODUCTION

RoboCup (Kitano *et al.*, 1997) is an international joint project to promote AI, robotics, and related fields. It is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined. RoboCup chose to use soccer as a central topic of research, aiming at innovations developed for soccer playing robots to be applied later for socially significant problems and industries.

CAMBADA (Almeida *et al.*, 2004*b*) is the Middle-Sized League soccer team from the University of Aveiro, and is composed of three field players and a goal-keeper. By itself the team is a dynamic multi-agent system with all players sharing their perception of the game field. Each robot is an autonomous unit, capable of making decisions in real-time based on its own sensorial data and from data received from its team mates. The software each robot runs is composed of several programs, all running simultaneously, taking large amounts of decisions in a very short period of time and performing complex tasks, which makes it very difficult to understand its reasoning while it's playing. Following the robot's reasoning based only on external observation is also difficult because it all happens very fast from the human point of view and most of the robots internal state is hidden. And we also need to consider that their decisions are based not only on information received from its own sensors but from the other team mates as well. This difficulty turns the process of tuning and debugging the decision mechanisms quite hard. To solve this problem a framework which uses the concept of layered disclosure (Stone *et al.*, 1999) and extends its functionalities was developed. It allows each process to log its data to a separate file and later join them to analyze the information saved as if it was on a continuous time-line. This process can be performed for each robot. A GUI program that allows navigating the files, searching for specific events and synchronize

video from the robots and other external sources was also developed.

This paper is organized as follows. This introduction is followed by section 2 with the main specifications that led to the development of the framework. Sections 3 and 4 discuss log creation and log navigation, respectively. Section 5 presents the GUI application developed for navigating the logs, synchronizing videos and searching for specific events. Section 6 introduces another tool for automatically determining the robots position in the field and help on the development of a self localization technique. Section 7 presents some results and finally on section 8 the conclusion of this paper.

2. SPECIFICATION

CAMBADA soccer team is composed of four players, three field players and a goal-keeper. Each robot operates autonomously processing the information obtained from the cameras, the base micro-controllers and also from the other team mates.

Fig. 1. CAMBADA Soccer Player

World information is shared between the robots using the RtDB TDMA protocol (Almeida *et al.*, 2004*a*). The purpose of the RtDB (Santos *et al.*, 2004) is to serve as both local and shared area for communication among processes within the same robot and communication among different robots and also to create a channel to communicate with the micro-controllers using FTT-CAN (Silva *et al.*, 2005). The RtDB is implemented using RTAI, a real-time layer for the Linux kernel.

Understanding what a particular robot is doing and why it is doing that is not easy, since it is a complex system, its world changes dynamically and it takes a lot of decisions per second. To help in this task we propose a debugging and monitoring system that simultaneously shows robot's knowledge and its real behavior. It is composed of a *back end* and a *front end*. The former is a library of functions and allows for the production of log files including video data. The latter is library of objects and a GUI application that allows the user to interact with single and multi-file logs from one or multiple agents. The tool allows for the synchronized visualization of the robot's reasoning through the contents of the log files, and of its external behavior, by displaying synchronously recorded video of the robot acting in the field.

2.1 Main Requirements

To create the framework to support the manipulation of log files several key points were defined.

First of all it should support generic text with information pertinent to the program. Second, it should be possible to organize the information by category, eg. *vision*, *decision*, *etc* and by level of detail so that when reading the log files, the user may have the ability to analyze specific parts of it.

CAMBADA robots run several processes at the same time so it's easier to create the framework so that it allows each program to write its own log file. Some form of synchronization is required to later be able to open them and read the information synchronously. This allows for the production of log files in different processes in the same machine, or even in different machines, and for overall analysis of the collected information. When applied to a soccer team this allows for:

- analyze together log data from different processes of a soccer player;
- analyze together log data from processes on different players;
- compare log data from a soccer player with data obtained with some monitoring system;
- analyze the robot's reasoning;
- analyze the real data on which robot's reasoning was generated.

Navigating log files and searching for specific information is another aspect to include in the specification. This means that some form of bookmarking is required for fast searching.

Recording video images is also required to be able to see what the robot sees at a given time and understand its decisions with the textual information.

Finally and probably the most important feature of the *back end* is the easiness of use so that other people will rapidly adapt to its interface and use it in their software.

2.2 Log structure

Since we will be logging multiple forms of information (text, image, bookmarking), we need to create different record types to record that information in the log files and to help navigating in them. So far we identified 4 types of records:

- *Text* to record formatted text messages;
- *Video* to save one image for example from the vision or from an external camera;
- *Bookmark* to place a bookmark for a specific category and later allow seeking on the information;
- *Registration* to register a new category of the tree to file;

The information contained in the logs may be overwhelming. On the other hand it should be as easy to visualize as possible. This has led to the use of classified information: vertically by level of detail; and horizontally by subject using a tree of categories.

To organize information by level of detail we use some points derived from the concept of Layered Disclosure as proposed by (Stone *et al.*, 1999), where the relevant information is organized in layers. Layers give us the depth of the information, that is, layers with smaller numbers indicate high-level reasoning of the robot and layers with higher numbers add more and more detail to the lower ones.

We extended the concept of layered disclosure with the inclusion of a tree of categories. This concept of tree was implemented on another library for logging system events (log4cpp (Bakker *et al.*, 2005)). The tree of categories is important to better organize the information, not only by its importance but by its type. Figure 2 shows an example of a tree of categories where it is possible to see the detailed structure of *run* (the agent of the robot) and *vision* which controls the front camera. The tree also gives the possibility to stop/start sending information of a given category to the log file in runtime or allow the person reading the files to hide categories that are not relevant for the analysis of the problem.

To be able to analyze multiple log files from one robot or from multiple robots, a common time line is required in all files. This can be done including a time-stamp in each record. Time-stamps can be obtained from the computer clock or from the RtDB/RTAI (Santos *et al.*, 2004). If we want to log data on multiple robots playing at the same time and later read it simultaneously, we need to synchronize their clocks. Synchronization can be done with NTP servers when using the computer clock as the source of time-stamps or from the RTAI layer in the Linux kernel.

Fig. 2. An example of a tree of categories with two programs running

2.3 Visualization of logs

Reading the log files and understanding the sequence in which the programs are executed is simple but it is highly time-consuming to do it by hand. This led us to create a second library to read log files simultaneously, interlacing them and give the person using it the impression that only one log file for each robot exists, the *front end*. This is done on-the-fly with all the selected log files and gives the user the exact order on which the programs ran. A GUI application was also developed in conjunction with the *front end* to allow navigation in the log and searching information by specific text, bookmarks, time, etc.

The framework includes the set of both developed libraries and the application to interactively analyze the log.

3. LOG PRODUCTION

CAMBADA robots run several processes in real-time simultaneously which makes impossible to use prints on the screen for debugging.

By using the monitoring framework's *back end*, every running program creates its own log file. The default file type is "*Formatted Text*" where every record is encoded in plain readable text, but other formats can be developed as well."*XML*" [1] or even a "*binary*" format are possible.

Every file is composed of records. There are four types of records created so far: *Registration*, *Text*, *Video*, *Bookmark*. The type of information each record contains is described in section 2.2.

Every record is composed, of:

- a time-stamp;
- a type;
- a category;
- specific data.

[1] Extensible Markup Language web site: http://www.w3.org/XML/

Here's an example of a registration and a text records:

```
35203551142 REGC /run/ctrlloop/decision/defender/
35203839343 TEXT /run/ctrlloop/integration/ball/ 2
    CORRIGIDA ws->ball is 1.86, 0.29
```

The first record has no specific data. It indicates that the program has just registered category /run/ctrlloop/decision/defender. The category being registered is **defender** but its full path along the tree of categories is written to the log file. The specific data of the second record is composed of a level of detail (2) and free text. It shows that the ball was seen at position (1.86, 0.29) relative to the robot's position and orientation.

To write these records to file and manage the tree of categories, several functions are available to the user and they are described in section 3.1.

3.1 System Architecture

When creating the *back end* library to write log files, one of the key aspects was to keep it as simple as possible to the user. So we decided the best is to present it as a library of functions in C.

Here is a small set of the most important functions available:

- `logInitialize` - given the file name initializes the tree of categories with the root category
- `logTerminate` - terminates all the logging facilities of the library, closes all files and releases all memory allocated
- `logText` - saves a text record
- `logBookmark` - saves a bookmark
- `logYuv` - saves a video image

Using categories is optional. An user may use only the root category which is created automatically by the library.

Using long strings, like the ones shown in the records example of section 3, to identify categories can be cumbersome and they are also prone to typing errors. To overcome this, an handle/descriptor number is created for each category and returned to the user by the `logRegisterNewModule` and `logRegisterSubModule` functions when new categories are registered. The handle may be used in subsequent calls to logging functions to identify categories.

These are the most important functions to manage the tree of categories

- `logRegisterNewModule` - registers a new category under the root of the tree and returns an handle
- `logRegisterSubModule` - registers a new category under any other existing category and returns an handle
- `logEnableOutput` - given the handle, enables logging of this category; it can also enable logging of all the subtree below this category
- `logDisableOutput` - given the handle, disables logging of this category; it can also disable logging of all the subtree below this category

Categories can be created and enabled/disabled on the fly when a program is running and can depend on conditions or program options, avoiding the need for recompiling. Figure 3 shows a simplified view of its components.

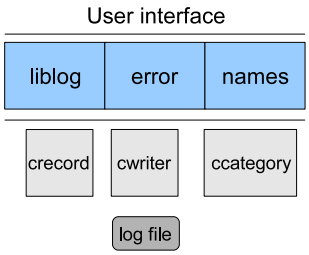

Fig. 3. User interface block diagram

Liblog is the main block. It implements the initialization of the library and the logging functions available to the user. *Names* implements the functions to manage the tree of categories. *Error* is a common block which makes available a set of functions to analyze errors of the library. *Error* is used by *Liblog* and by *Names* to manage errors internally and can also be used be the user to print or analyze them. All functions provided by these blocks are developed using dynamic buffers to prevent suspending the process execution and maintain the impact on the performance to a minimum. Also, all functions have a small number of parameters and some of them are similar to system functions, making them very easy to use.

The remaining blocks: *crecord*, *cwriter* and *ccategory* are support classes for the objects used internally by the library: different record types, formatted text output and each category in the tree, respectively. This means that the core of the framework is built in C++ because it is simple to write code that closely represents the conceptual model of the project using an Object Oriented programming language. The *back end* interface is a wrapper to these objects in C so it will be simple to use. To add new functionalities to the framework like new types of records or new log file formats it's as easy as creating new derived classes of these.

4. LOG NAVIGATION

4.1 Synchronization

To read data from different log files and retrieve joined information a kind of synchronization is required. This synchronization is based on time-stamps included in each record, on every file. Rebuilding the original sequence on which the records were saved to the log files is possible just by implementing an algorithm to seek the record with the closest time-stamp to the current one.

Several classes that represent the *front end* of the framework are implemented as seen in figure 4, using C++ and the STL library to maintain the portability between multiple platforms. They implement gradually several forms of navigating the information:

- **CFile** - Implements basic file I/O, open, read and seek functionalities;
- **CParser** - Implements file interpretation, creates records from a file and allows sequential navigation on them and has some caching built-in;
- **CTimeNavigator** - Extends the CParser functionalities by adding the ability to navigate using time-stamps (seek to time-stamp) besides the already existing sequential navigation;
- **CCursor** - Uses all the functionalities of the classes above and a look ahead technique to allow sequential navigation and by time-stamp on multiples files at the same time;

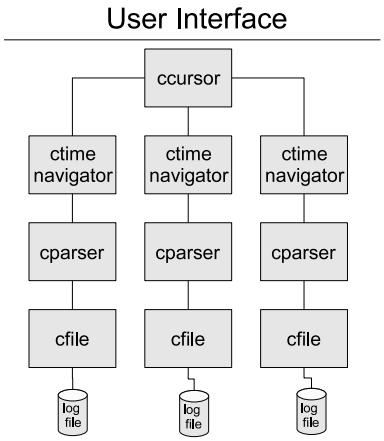

Fig. 4. User interface block diagram

4.2 Multiple log file navigation

CCursor is the top level class for this *front end* library of the framework. Its name comes from the analogy of a sliding cursor used to navigate all the records on multiple files. It can manipulate multiple log files at the same time and synchronize them to the current time-stamp. Current time-stamp can be interpreted as the current position of the cursor.

CCursor makes available to the user, the time-stamp limits for the set of files and the ability to seek to a specific position and navigate from there, forward or backwards.

The tree of categories can be created by the user from the log files managed by CCursor by means of CCategory objects and CCategoryIterator iterators which implement *depth-first search* algoritm.

Based on CCursor, it was possible to develop the GUI application (section 5) that receives the records already "organized" so they can be filtered and displayed according to their nature, either text or video.

5. LOGREADERQT APPLICATION

To make practical use the log files in CAMBADA to debug the robot's software a GUI application was developed using the Qt framework (Trolltech, 2005). An image of the main window is shown in figure 5. So far this application has two modes of displaying information, one textual and one with video.

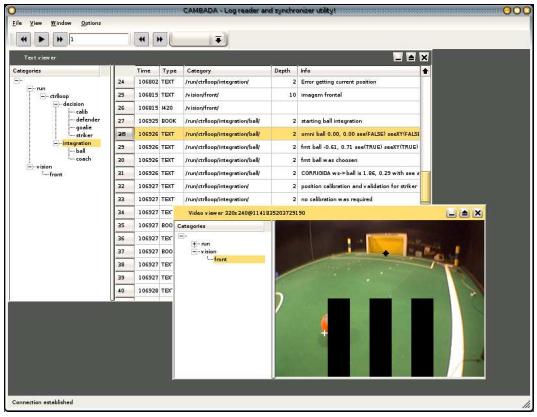

Fig. 5. Logreader main window

When LogreaderQt starts it creates a CCursor object and an empty tree of categories. The CCursor object will maintain and manage the set of log files provided by the user and it will also allow navigating the records to extract information to display and to create the tree of categories. Log files can be added and removed from the set whenever it is required.

Navigation in the set of files is entirely done by CCursor object. Figure 5 shows the tree of categories, inside each window, that is created when CCursor opens files. It is possible to create empty categories on the tree and to "mount" log files in them, using the same analogy of a file

system. This allows to separate logs from different agents and applications if required.

Filtering details and/or hiding data is possible using context menus either for *categories* of data or for *level of detail* of information. This way an user can hide everything that is not important and read only what matters most for a particular analysis.

LogreaderQt gives the user several ways of searching for what he's looking for in the log. It is possible to search information using:

- **records** - seek a number of records at a time and showing them;
- **time** - seek some time forward or backwards in the log. The time units for the files and navigation are user selectable;
- **bookmarks** - seek inside the log to the next or previous bookmark of the selected category; this way it's possible to jump from one control cycle to the next for example;
- **video** - seek to the next or previous image of the selected category;
- **play** - play mode to keep advancing the records and video until the user instructs the program to stop;
- **regular expressions** - in the future;

Finally video and text are synchronized, which means when a text record is selected the video jumps to the nearest previous image. This simplifies the process of analyzing information by the user as it allows the direct comparison of robots internal state and reality as seen by the camera.

6. REAL POSITION MONITORING

One of the problems we faced while developing the soccer team is the fact that the robot's absolute position on the field is not trustable. Moving along the field while playing, when the robot's absolute position is updated only by odometry it tends to accumulate errors after travelling a few dozen meters.

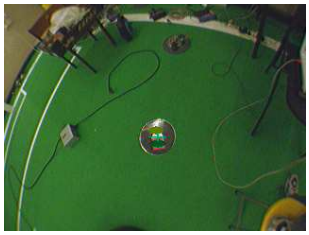

Fig. 6. Top viewer detecting a robot top marker

Facing this problem we devised a solution to help quantify the error of the current solution and test new solutions as they are implemented. Like the Robocup's Small-Sized League, we incorporated a video camera on the top of the game field. This camera has its own software, figure 6, to identify the field limits, the robotic agents in the field and log their position (text and video) to a file, using the framework. Comparing this log and the robots' logs should give us the ability to identify current problems and evaluate new solutions.

7. RESULTS

To demonstrate a typical application of the framework, log files were created on two robots that exhibited a strange behavior while playing soccer. The setup for the experience consisted on leaving the robots side by side and placing the ball at a distance were the problem was visible. It was clear that if they seen the ball simultaneously and if their distance to the ball was similar at that time, robot number 3 was always the one to change to striker.

After opening the log files with the application it became clear were the problem was coming from. Figure 7 shows one of the images recorded by the robot's log. Figure 8 shows an excerpt of textual information from the log files. There we can see that robot number 3 detects the ball at a distance of 4 meters and robot 1 at 6 meters. Clearly robot 3 thinks it is the closest to the ball and changes its behavior to striker. The source of the problem is in the neural network that translates pixels from the image to distances in the field not being properly calibrated. After analyzing the rest of the log we also discovered that the vision of robot 3 was not returning any distances greater than 5 meters.

Fig. 7. Log image from the vision

27	1338714	TEXT	/vision/fball1/	1	fball x y ang -- conf: -0.20 6.03 1.60 -- 1.00
28	1403892	TEXT	/vision/fball3/	1	fball x y ang -- conf: -0.72 4.19 1.74 -- 1.00
29	1405307	TEXT	/vision/fball1/	1	fball x y ang -- conf: -0.13 6.03 1.59 -- 1.00
30	1470459	TEXT	/vision/fball3/	1	fball x y ang -- conf: -0.72 4.19 1.74 -- 1.00

Fig. 8. Excerpt from the textual information

Like this problem, that was quickly discovered, many others can be easily spotted with the ability to cross information from multiple processes and multiple agents simultaneously if we use these tools.

8. CONCLUSION

In this paper we presented a framework developed for debugging a robotic soccer team which proved to be useful for a human user to understand the reasoning of an agent. Although it has been developed for soccer, its implementation has been carefully done to allow it to be easily adaptable to other projects where off-line debugging of multiple autonomous robots with different categories of information is required.

It's main capabilities were easiness of use on single and multi-agent systems, multiple log files per agent (reading and writing), information organization in categories and level of detail, synchronization with video and different methods of searching information in the log.

REFERENCES

Almeida, Luís, Frederico Santos, Tullio Facchinetti, Paulo Pedreiras, Valter Silva and Luís Seabra Lopes (2004*a*). Coordinating distributed autonomous agents with a real-time database: The cambada project.. In: *ISCIS*. pp. 876–886.

Almeida, Luis, Luis Seabra Lopes, P. Bartolomeu, E. Brito, M. B. Cunha, J. P. Figueiredo, P. Fonseca, C. Lima, R. Marau, N. Lau, P. Pedreiras, A. Pereira, A. Pinho, F. Santos, L. Seabra Lopes and J. Vieira (2004*b*). CAMBADA: Team Description Paper. In: *CD of the Robocup Symposium / TDP*.

Bakker, Bastiaan, Cedric Le Goater, Marc Welz, Lynn Owen andSteve Ostlind, Marcel Harkema, Uwe Jger, Walter Stroebel, Glen Scott, Tony Cheung, Alex Tapaccos, Brendan B. Boerner, Paulo Pizarro, David Resnick, Aaron Ingram, Alan Anderson and Emiliano Martin (2005). *Log for C++ Project Website*. 0.3.5rc3 ed.

Kitano, Hiroaki, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda and Eiichi Osawa (1997). RoboCup: The Robot World Cup Initiative. In: *Proceedings of the First International Conference on Autonomous Agents (Agents'97)* (W. Lewis Johnson and Barbara Hayes-Roth, Eds.). ACM Press. New York. pp. 340–347.

Reis, Luís Paulo and Nuno Lau (2000). *FC Portugal Team Description: RoboCup 2000 Simulation League Champion*. pp. 29–40. Vol. 2019. Springer-Verlag.

Santos, Frederico, Luis Almeida, Paulo Pedreiras, Luis S Lopes and Tullio Facchinetti (2004). An Adaptive TDMA Protocol for Soft Real-Time Wireless Communication among Mobile Autonomous Agents. *WACERTS'04 RTSS'04*.

Silva, Valter, Ricardo Marau, Luís Almeida, J. Ferreira, M. Calha, P. Pedreiras and J. Fonseca (2005). Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In: *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*. Vol. 2. pp. 781–788.

Stone, Peter, Patrick Riley and Manuela Veloso (1999). Layered Extrospection: Why is the agent doing what it's doing?. *Fourth International Conference on Autonomous Agents (Agents-2000)*.

Trolltech (2005). Trolltech - Cross-platform C++ GUI development Online Reference Documentation. In: *http://doc.trolltech.com/*.