Design and Implementation of a NetLogo Interface for the Stand-Alone FYPA System

Daniela Briola and Viviana Mascardi
Dipartimento di Informatica e Scienze dell'Informazione (DISI)
Università degli Studi di Genova
Email: {briola, mascardi}@disi.unige.it

***Abstract*—FYPA (Find Your Path, Agent!) is a multiagent system currently used by Ansaldo STS for off-line daily computation of paths of trains inside stations. Its exploitation for on-line re-planning in case of unavailability of resources is envisaged in the very near future, since the system's performances demonstrated to be suitable for real time usage.**

In this paper we present StandaFYPA, the stand-alone version of FYPA that we developed for running batteries of tests on our own, without needing to access existing Ansaldo applications. StandaFYPA is equipped with a graphical interface implemented in NetLogo for off-line visualization, that we describe here in details.

I. Introduction

FYPA (Find Your Path, Agent!) [2], [4], [5] is a multiagent system used by Ansaldo STS, the Italian leader in design and construction of signaling and automation systems for conventional and high speed railway lines, for solving a resource allocation problem by means of distributed negotiation.

The system is currently used by Ansaldo STS for off-line computation of paths of trains inside stations, whereas its on-line application for real-time unavailability management is foreseen in the very near future.

The "Ansaldo FYPA" system (AnsaFYPA from now on), namely the system currently operating in Ansaldo STS centers, reads configuration data from a legacy system and send results to it, using Web Services. In 2009 we started the development of a stand-alone version of the system in order to carry out batteries of tests without needing any Ansaldo legacy system, whose access from outside Ansaldo was of course not allowed. Hence, we implemented the "Stand-alone FYPA" system (StandaFYPA in the sequel) by reusing as much as possible code we already developed for Ansaldo, and by implementing the new functionalities we needed for making StandaFYPA self-contained.

AnsaFYPA and StandaFYPA are different only as far as the input/output management is concerned: we changed the type source of input data and we modified the output visualization. In particular, we added a post-execution simulator to the StandaFYPA system implemented by means of a NetLogo [6] program able to read the output files of the agents and to graphically represent the moves of trains (agents) on the railway tracks (resources).

The performances of StandaFYPA and its NetLogo interface are the main subjects of this paper, which is organized in the following way: Section II briefly recalls the problem addressed by FYPA and the proposed solution, and provides an evaluation of AnsaFYPA and StandaFYPA performances; Section III provides some background knowledge on NetLogo, discusses the reasons why we did not implement StandaFYPA in NetLogo, and shows how we used NetLogo for developing the StandaFYPA graphical interface; Section IV shows StandaFYPA at work; finally, Section V outlines some future directions of research and concludes.

II. Stand-alone FYPA

A. The problem faced by FYPA

The abstract problem faced by the FYPA multiagent system has already been described in [2], [4], [5] and consists of

- A set of indivisible resources that must be assigned to different agents in different time slots (each resource can be used by only one agent in each time slot).
- A set of agents with different priorities, each needing to use some of the available resources for one or more time slots; agents have preferences over the set of resources they can obtain.
- A directed graph of dependencies among resources: an agent can start using resource R only if it used exactly one resource from $\{R_1, R_2, ..., R_n\}$ in the previous time slot (we represent these dependencies as arcs $R_1 \rightarrow R$, $R_2 \rightarrow R$, ..., $R_n \rightarrow R$ in the graph).
- A set of resources named "start points" that can be assigned to agents without requiring the prior usage of other resources (no arc enters in the corresponding node).
- A set of resources named "end points" that, once assigned to one agent, allow the agent to complete its job (no arc exits from the corresponding node).
- A set of couples of conflicting arcs in the graph of dependencies: an agent releasing R_1 for accessing R_2, where the usage of R_2 depends on the previous usage of R_1, might conflict with an agent releasing R_3 for accessing R_4. The two agents might indeed need to use the same transportation means for accessing R_2 from R_1 and R_4 from R_3 respectively, and the transportation means might be non sharable as well.
- A static allocation plan that assigns resources to agents for pre-defined time slots, in such a way that no conflicts arise.

Since agents happen to use resources for longer than planned and resources can break up, a dynamic re-allocation of resources over time is often required. Thus, the solution of the real world problem is a dynamic re-allocation of the resources to the agents such that:

1) the re-allocation is feasible, namely free of conflicts; in our scenario, conflicts may arise both because two or more agents would want to access the same resource in the same time slot, and because two or more agents would want to use conflicting arcs in the same time slot;
2) the re-allocation task is completed within a pre-defined amount of time;
3) each agent minimizes the changes between its new plan and its static allocation plan: the start and end point must always remain those stated in the static allocation plan, but the nodes in between may change, as well as the time slots during which resources are used;
4) each agent minimizes the delay in which it reaches the end point with respect to its static allocation plan;
5) the number of agents and resources involved in the re-allocation process is kept to the minimum.

In FYPA, every train is managed by a "Treno agent" and every resource inside the station (a node of the graph) is managed by a "Nodo agent"[1]. Railway tracks connecting nodes (the arcs of the graph) are resources to be assigned to trains. The resource allocation problem is solved by means of a complex negotiation protocol that converges towards a solution provided that

- there are always more free resources than agents;
- at least one complete free path connecting every start point to an end point exists;
- the number of arcs in the graph ensures a redundancy in the choice of paths;
- agents enter the graph at the time stated by their static allocation plan or later;
- it cannot occur that two trains reserve the same resource in a "not disputable" way.

Both AnsaFYPA and StandaFYPA solve the above problem. The table below summarizes the differences between the two systems, which are concerned with technical details of their implementations and not with the functionalities they provide.

	AnsaFYPA	**StandaFYPA**
OS	Linux	Windows (XP or Vista)
Input	Text Files	Database
Data Interf.	Web Services (WSIG)	Connection to DB
Paths	Saved in an external file	Computed run time from DB
Arcs	Mainly One-Way	Bidirectional
GUI	Ansaldo STS program	NetLogo

B. *Evaluation of the AnsaFYPA and StandaFYPA systems*

AnsaFYPA demonstrated to give good performances when tested by Ansaldo STS engineers: using a station with about fifty nodes, two hundreds arcs and at least ten trains in the station at the same time, the system was able to find a solution in few seconds. Ansaldo STS is using AnsaFYPA as a plug-in of its commercial application, and uses it to organize the movements of trains in a station for an entire day. In the sequel we report data of two experiments carried out by Ansaldo STS engineers on the field (courtesy of Ansaldo STS).

The first station considered during the testing of the AnsaFYPA system was Mestre: this station has 59 Nodo agents, 528 Treno agents (during day 28th of March 2011) and each Nodo agent manages approximately tree entering arcs. The total number of incompatibilities is 430. A simplified representation of Mestre station is shown in Figure 1.

The simulation carried out by Ansaldo STS engineers took 18 minutes to be completed. This time is also due to the scheduling chosen by the Ansaldo STS system: the User Agents Manager creates a new train every 2 seconds, so the simulation time is at least equal to the number of trains multiplied for 2 seconds: in this example, at least 17 minutes. This means that the system requires less than two seconds, considering all the simulation time, to manage the reallocations/conflicts that arise between trains when a new train enters the station. This amount of time is acceptable for using AnsaFYPA on-line as well.

The second station that was used for testing AnsaFYPA is Pisa. Pisa is managed by 60 Nodo agents, each having about 20 entering arcs. Every arc has 130 incompatibilities on average. This station is definitely more complex than Mestre because of the many more incompatibilities. The simulation of the real traffic in Pisa station on the 20th of January 2011, with 395 trains crossing the station, took only 13 minutes to be completed (with one train entering the station every 2 seconds, like in the previous example). A graph-like representation of Pia station is reported in Figures 2 and 3, where the station is partitioned into its left and right sub-graphs.

The tests we carried out on StandaFYPA show that it performs in the same way as AnsaFYPA if considering the "allocation phase", that is, if considering the simulation starting when the first train enters the station. StandaFYPA is faster than AnsaFYPA in its "set up" phase (the phase where Nodo agents are created with the information regarding arcs and incompatibilities) because the direct access to a database is definitely faster than using web services and parsing very large files. The StandaFYPA system takes about thirty seconds to set up, while the AnsaFYPA needs some minutes.

III. NetLogo and its usage in Stand-alone FYPA

A. *NetLogo*

This section provides a short introduction to NetLogo and is based on [1] and on http://ccl.northwestern.edu/netlogo/docs/.

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of agents all operating independently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals.

[1]Being developed for an Italian company, FYPA code uses Italian tags for agent names: in this paper we keep the names of agents as they are in the code, to avoid inconsistencies with respect to the images captured from JADE.

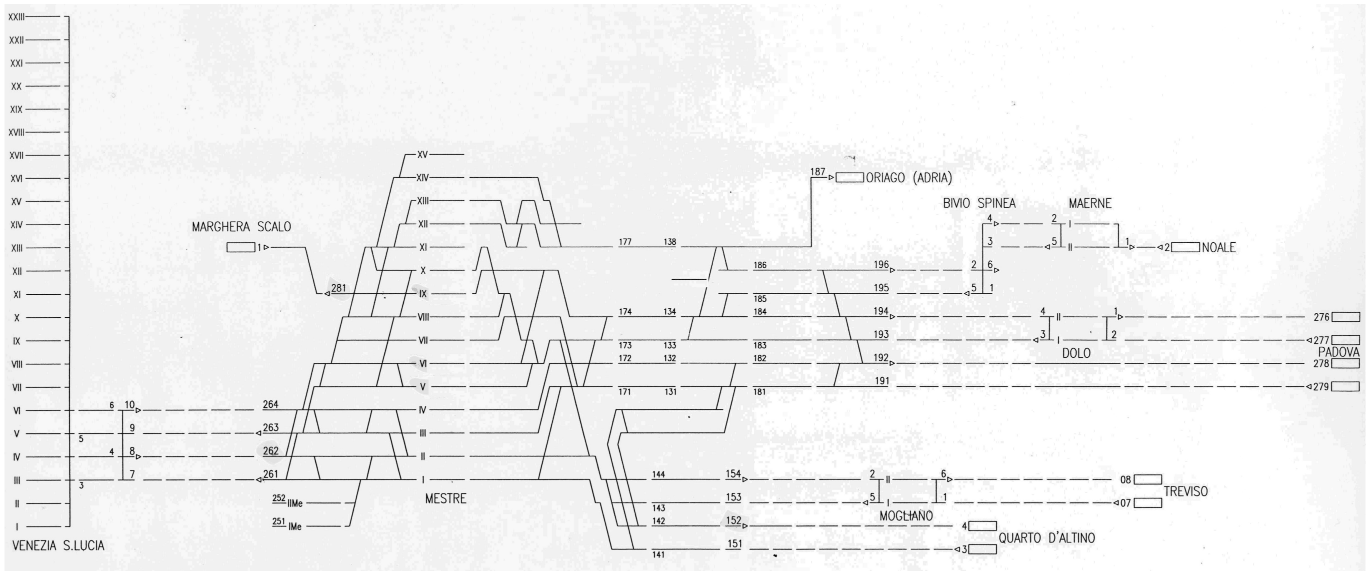

Figure 1. The simplified representation of Mestre station (courtesy of Ansaldo STS)

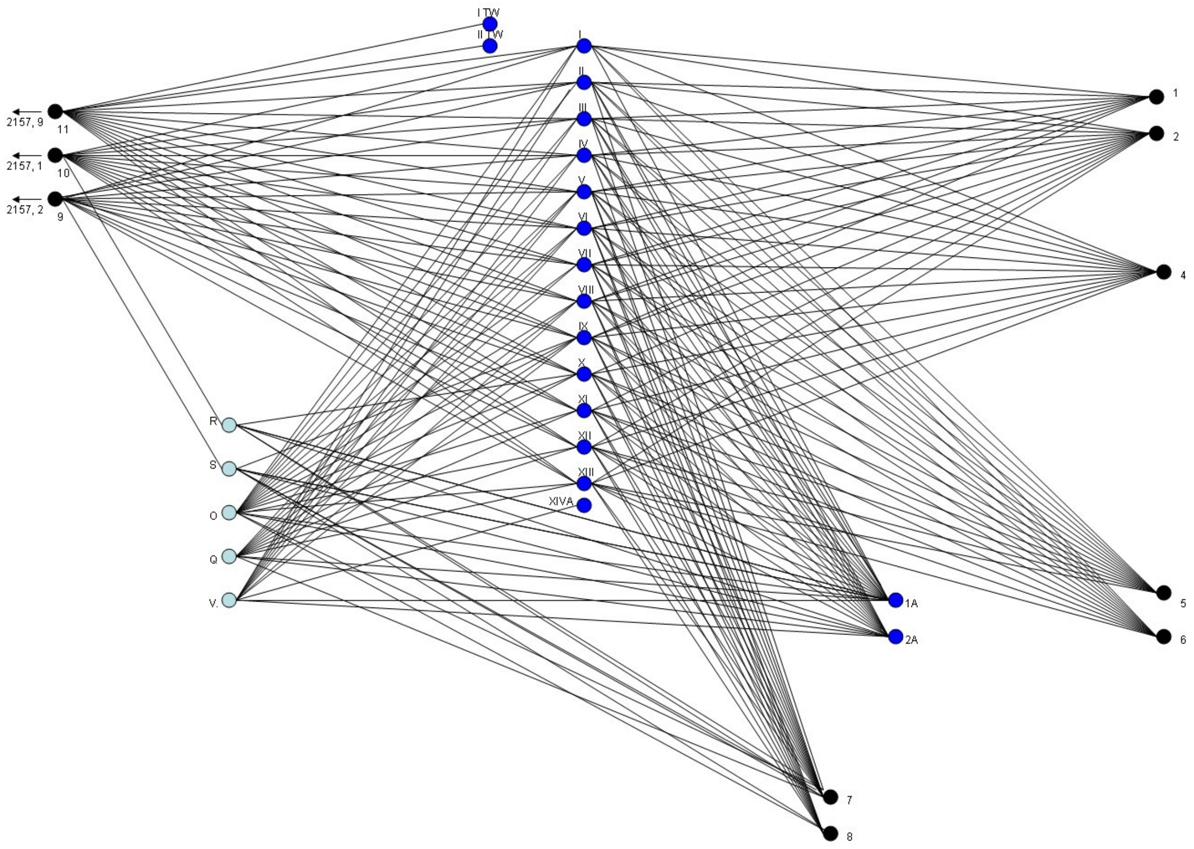

Figure 2. The graph-like representation of Pisa station (right side, courtesy of Ansaldo STS)

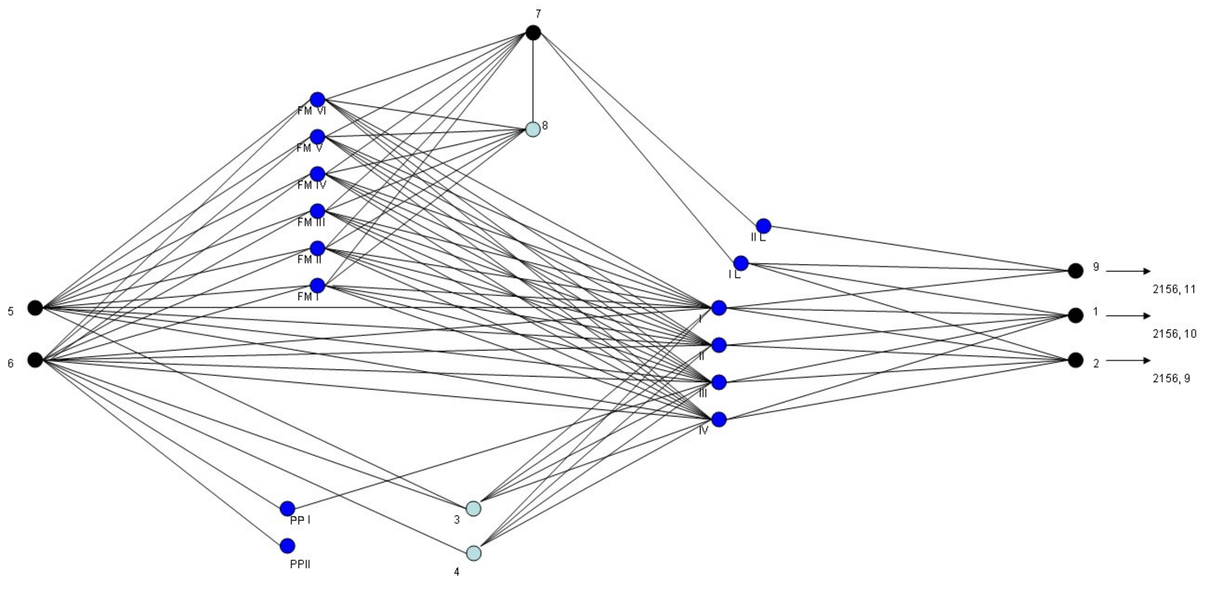

Figure 3. The graph-like representation of Pisa station (left side, courtesy of Ansaldo STS)

NetLogo supports three types of agents: turtles, patches, and links. Patches represent square (in 2D) or box (in 3D) cells on the main 2D (or 3D) view of the world. Turtles are agents that can move around on the world surface, and draw. Links represent relationships between turtles. There is also an "observer agent" that has a view of the whole NetLogo world and is used for running the main parts of the program (linked to buttons on the interface) as well as providing a way of interacting command by command on the main interface.

One of the benefits of using NetLogo, which is in fact the reason why we used it in StandaFYPA, is its graphical interface. By default, the interface contains just a 2D spatial view of the model environment, which is a square lattice. In addition to the 2D spatial view, the developer can add other elements such as buttons to set model parameters and graphs to monitor results. The 2D view has several different options that prove useful in modeling. The size of the grid (the number of cells) can be changed, as can the size of the cells themselves (in pixels).

B. Why not implementing StandaFYPA in NetLogo

NetLogo is very useful to simulate the evolution of a system consisting of thousands of simple agents divided into different categories. Hence, it is suitable to implement applications whose main aim is to evaluate the emergent behavior of the system and where there is no need to follow the life cycle of individual agents. In that kind of applications, decisions about what action is to be done are usually made using a probabilistic choice. Also, simulated time does not require a sophisticated management and the built-in representation of time provided by NetLogo by means of Ticks (a discrete representation of time shared among agents) is enough.

We could not use NetLogo to implement the stand-alone version of the FYPA algorithm mainly because of the following reasons:

1) NetLogo is not suitable to simulate a negotiation among agents based on a "deterministic view" (namely, a negotiation based on a protocol with predefined roles and actions);
2) NetLogo is not suitable to simulate an interaction where agents exchange messages;
3) NetLogo does not support a continuous model of time we adopted in FYPA.

Besides these main and almost general motivations, there are more specific ones due to the features of the physical world where our agents (trains and nodes) live, and to the protocol we designed.

The environment of our MAS can be represented as a graph with nodes and many arcs connecting them, and agents that cross the graph. In NetLogo, links represent a connection between two agents, but only one link between each couple of agents is allowed: the existence of multiple railway tracks among two nodes was difficult to model. Furthermore, the most natural way of using NetLogo is to ask the entire population of agents of some kind to do something, using the command "ask *agentset* to do something": this command selects one agent from the set in a random way and than executes the code for that agent. Then it randomly chooses another agent and so on, until all the agents of that kind have been selected. In this way, the execution of the entire protocol is forced to be sequential, and not simultaneous as in FYPA happens, and hence all the problems arising from the concurrent choice of a resource (path, arc or node) do not emerge and cannot be dealt with.

Although there are ways to avoid this sequential behavior of the "ask command", and hence this last problem should have been overcome, we soon realized that, in order to use NetLogo to implement StandaFYPA, we should have forced NetLogo to behave in a completely different way with respect to its own philosophy. Instead, NetLogo proved a very suitable tool for implementing a nice graphical interface for off-line visualization of train paths: we used it for exploiting this functionality, as described in the next section.

C. NetLogo graphical interface for StandaFYPA

NetLogo offers an integrated graphical representation of the simulation, so it is almost simple to let the user see how the simulation is going on. Instead of programming the graphical front end of an application, to see for example where the agents are moving, one can use NetLogo that provides these visualization facilities for free.

We took advantage of them for off-line visualization of the StandaFYPA output. Our NetLogo program reads the structure of the station and the movements of a set of Trains during the time from the log files that are output by StandaFYPA (manually reworked to meet the input format needed by our NetLogo program). Due to NetLogo limitations, we are not able to draw more that one arc between two nodes.

Nodes and trains are NetLogo turtles, whereas arcs are NetLogo link agents. Time is represented using NetLogo $Ticks$: at every tick, each train will read from a private list (initialized with its movements on the graph) and will move on a new node (or will remain on the node where it is, if it should stay there).

NetLogo needs four types of files:

- Nodes.txt: a list of nodes with their physical position on the NetLogo output screen;
- Arcs.txt: a list of the arcs connecting two nodes (one arc for each pair at most);
- Trains.txt: list of all the trains involved in the simulation;
- Movements.txt: list representing the position of trains on nodes, for each NetLogo time tick.

Figure 4 shows the interface of our NetLogo program. When the program is started, all the agents read the information they need from these four files. The user can decide to run the simulation "tick by tick" or "as a movie" (selecting the flag "Forever" among the options of the "Go" button): in the first case, the user will ask the system, pressing the button "Go", to move on the simulation of only one tick (updating the trains position), while in the second case the system will update the graph every $Delta$ seconds (that is, every $Delta$ seconds the tick counter will be incremented and the position of trains updated), showing in this way the trains moving on the graph. $Delta$ is a value that the user can change to slow down or speed up the simulation, using the sliding bar shown at the top of the interface.

IV. StandaFYPA at work

In this section we will describe the main features of StandaFYPA using some examples: we will start with simple ones to show the basic interactions in the negotiation protocol, how the changes of the state of the graph are spread among Nodes, the node reservation procedure and so on. Then we will present some more complex examples (using sometimes a Dummy agent from JADE platform) to show the interactions among Trains and Nodes.

To fully understand the examples we should recall the FYPA system uses Italian names for agents, in particular:

- User Agents (UA) are called "Treno"
- Resource Agents (RA) are called "Nodo"
- Resource Agents Manager is called "Prenotazioni-Stazione"
- User Agents Manager is called "MovimentiIndotti"
- Paths Manager Agent (PA) is called "PercorsiAlternativi"

These examples have been executed on a personal computer with MS Windows XP Professional$^{©}$, 2 GB RAM and an AMD Athlon$^{©}$ 64 processor 3000+.

A. Example 1: Resource reservation

In this example we show how Nodo agents interact to maintain the state of the graph updated.

Considering the graph shown in Figure 5, $Nodo\_3$ and $Nodo\_4$ manage arc number 6 which has an incompatibility with arc 5, managed by $Nodo\_2$ and $Nodo\_5$. These arcs are managed (namely, "in the scope of") by two Nodo agents because they are bidirectional.

Let us simplify the example using a Dummy agent ($da0$) instead of a Train and let us suppose that this agent needs to leave $Nodo\_3$ to move on $Nodo\_4$. In this case $da0$ will send a reservation request to $Nodo\_4$ and then a confirmation message to it.

In Figure 6 the messages exchanged between Nodo agents are reported: $Nodo\_4$ informs $Nodo\_2$, $Nodo\_3$ and $Nodo\_5$ that there is a reservation request they must be informed of, and then it sends a new message, with performative CONFIRM, to inform its neighbors that the previous reservation request has been confirmed.

The execution of this example in JADE takes less than 1 second to be completed.

B. Example 2: Path reservation, with free nodes

In this example we describe the procedure to reserve a path, and we start from the simplest situation, that is, all the nodes are free and there is only one train in the station.

In the table below, the original plan of $Treno\_1$ is reported: considering the station shown in Figure 5, $Treno\_1$ needs to reserve the path *Nodo_1, Nodo_3, Nodo_4 and Nodo_6*. In Figure 7 the complete interaction between $Treno\_1$ and nodes is shown.

Train	**Step**	**Node**	**From (ms)**	**To (ms)**
Treno_1	1	Nodo_1	210000	240000
Treno_1	2	Nodo_3	240000	310000
Treno_1	3	Nodo_4	310000	340000
Treno_1	4	Nodo_6	340000	380000

In Figure 8 the movements of $Treno\_1$ in the station are reported: these images come from the NetLogo interface described in Section III-C. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 1 second to be completed.

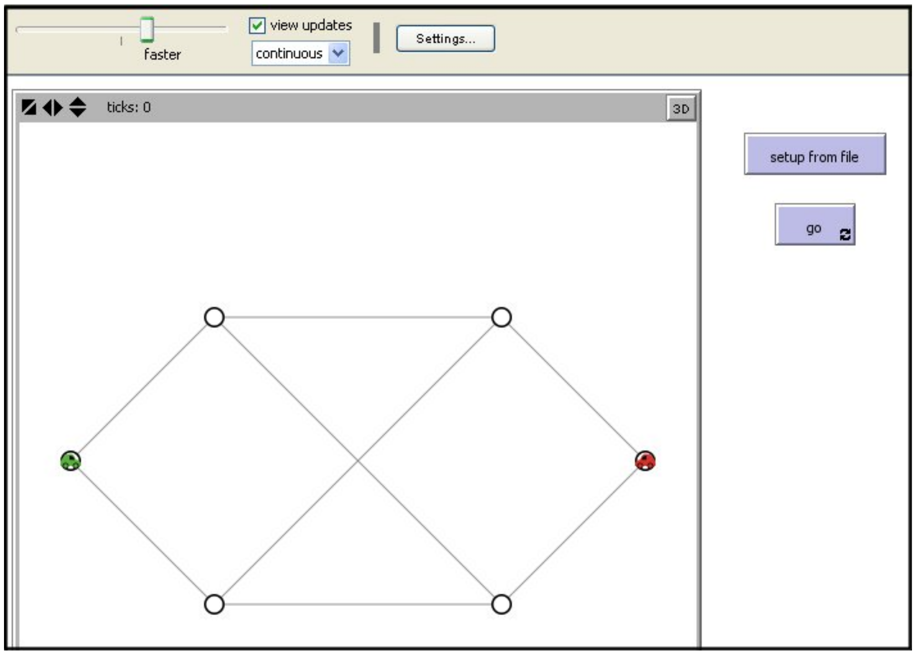

Figure 4. StandaFYPA graphical interface implemented with NetLogo

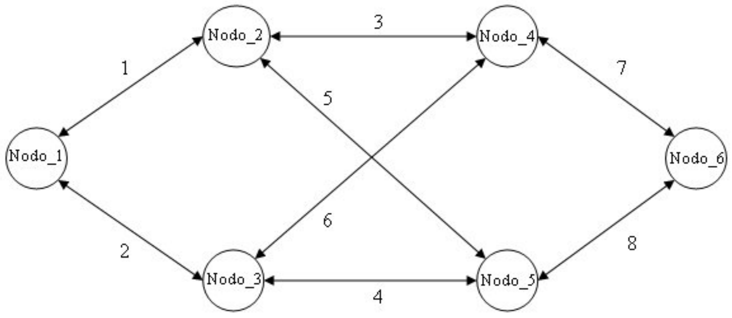

Figure 5. Station represented as a graph

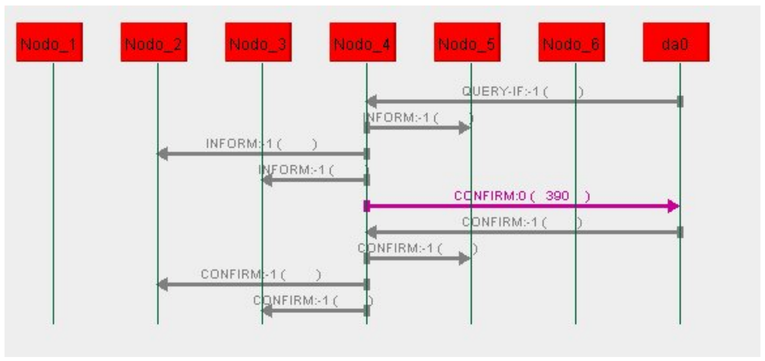

Figure 6. Example 1: JADE sniffer view

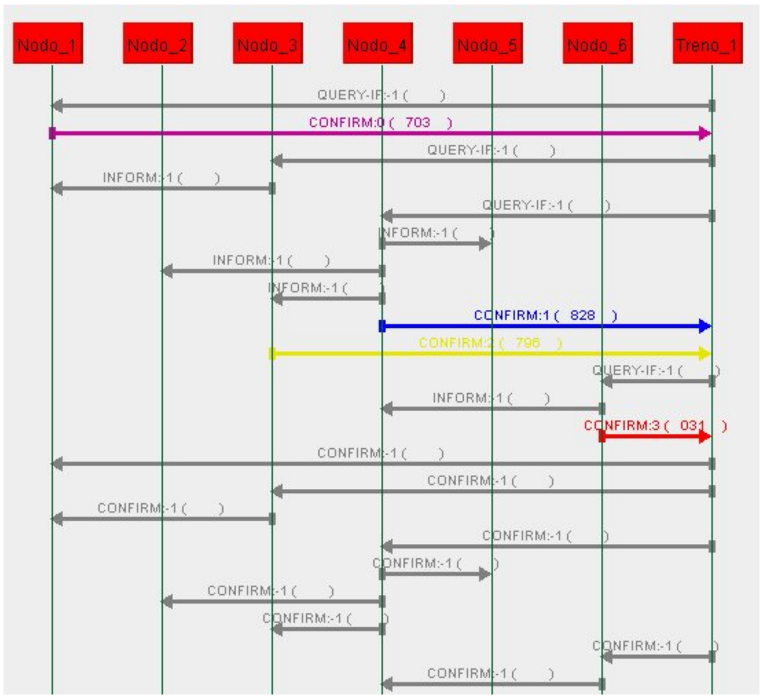

Figure 7. Example 2: JADE sniffer view

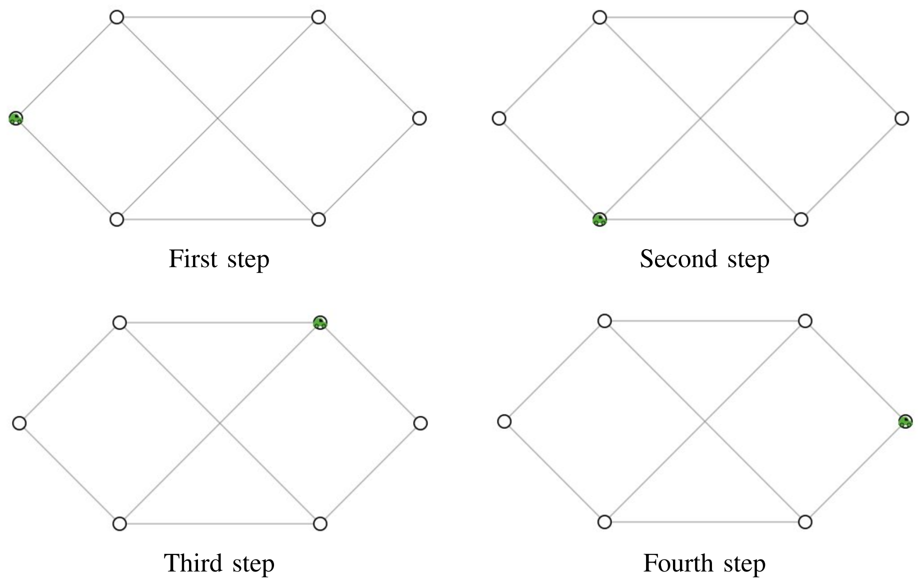

Figure 8. Example 2: NetLogo screenshots

C. Examples 3 and 4: Path reservation with occupied nodes

Now let us consider a further more complex situation that we label with "Example 3". Suppose that $Treno\_1$ has priority equal to 2. The station and $Treno\_1$'s original plan are the same ones of the above example.

We will show how the Treno agent acts if one of the resources it is trying to reserve is already occupied.

In this example we assume that another train (a dummy agent, $da0$) has already a reservation for the resource $Nodo\_4$ for the interval 310000 - 340000 (that is the same interval requested by $Treno\_1$). The behavior of $Treno\_1$ will change considering the maximum delay it can undergo and the priority of $da0$.

Let us suppose that $Treno\_1$ can now accept 4000 milliseconds as maximum delay, and that $da0$ priority is 3 ($da0$ is a train with a lower priority than $Treno\_1$).

As the reader can see in Figure 9, when $Treno\_1$ sends it QUERY-IF messages, it receives an INFORM from $Nodo\_4$ that specifies that the resource is already occupied by train $da0$, with priority 3, and that the resource will be again available from 340001 (till 370001, that is, is free for the same time interval but starting from 340001).

$Treno\_1$ can calculate the total delay it should accept: in this case, 3000 milliseconds, that is an acceptable delay. The train decides to maintain its path (even if it could steal the resource from $da0$) shifting the reservations: it asks $Nodo\_3$ to stay more on it, and then it moves on $Nodo\_4$ when specified in the INFORM message. All the nodes will answer with a CONFIRM, so the train will again confirm its reservations and will get a reserved path.

The interaction among the agents (excluding the last CONFIRM messages sent back by $Treno\_1$) is shown in Figure 9, while the complete simulation made with the NetLogo interface is reported in Figure 10. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 2 seconds to be completed.

In this situation, the priority of $da0$ does not care because $Treno\_1$ can accept the delay suggested by $Nodo\_4$.

However, let us suppose that the dummy agent $da0$, that has already a reservation for the resource $Nodo\_3$ for the interval 240000 - 310000, has priority 3 and that the maximum delay for $Treno\_1$ is 1000.

In this example $Treno\_1$ steals the resource from $da0$ because it has an higher priority and the delay it should accept shifting its reservations is too high (we do not provide screenshots for this example).

An even more complex situation, that we label with "Example 4", takes place if the dummy agent $da0$, that has already a reservation for the resource $Nodo\_4$ for the interval 310000 - 340000, has priority 1 and that the maximum delay for $Treno\_1$ is 1000.

Being 3000 milliseconds of total delay not acceptable and being the priority of $da0$ higher than the one of $Treno\_1$, the train can only search for another path. It asks the Paths Manager Agent (PA) to get the list of alternative paths (from $Nodo\_1$ to $Nodo\_6$), avoiding $Nodo\_4$. The content of the QUERY-IF message sent by $Treno\_1$ to PA is the next one:

1; $Nodo\_1$; $Nodo\_6$; $Nodo\_4$; [$Nodo\_1$, $Nodo\_3$, $Nodo\_4$, $Nodo\_6$]; $Nodo\_4$

while the content of the INFORM message received from PA is the next one:

$[(1\_false; 2\_true; 5\_true; 6\_false),$
$(1\_false; 3\_true; 5\_true; 6\_false)]$

that is the list of alternative paths (where 1_false means $Nodo\_1$ that is not a "Stop Node"). In this case, $Treno\_1$ selects the path $(1\_false; 2\_true; 5\_true; 6\_false)$ and it succeeds in reserving it, as shown in Figure 11. In Figure 12 the complete simulation made with the NetLogo interface is reported. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 2 seconds to be completed.

V. Conclusions and Future Work

In this paper we discussed the StandaFYPA systems, its differences with respect to AnsaFYPA, its performances on real case situations (Pisa and Mestre stations) and its behavior in small examples that we used for debugging purposes.

We already started to look for similar case studies in the same area to verify if StandaFYPA may be used to solve them [3]: by analyzing these new case studies in detail and trying to solve them using our StandaFYPA system we will also be able to understand how our protocol can be modified to cope with them and, in general, how it can be made more flexible.

Some technical improvements that are on their way are:

- implementing the code to automatically export StandaFYPA log files in the format requested by our NetLogo graphical interface (currently, some manual adjustment is required);
- modifying the database (and the code) to model directed arcs instead of bidirectional arcs.;
- modifying the protocol to let Treno agents "ask help": if a Treno agent $T1$ needs a resource that is already reserved by another Treno agent $T2$ with an higher priority (so $T1$ is not able to steal the resource), $T1$ could ask $T2$ to find another path in the station, that is, it could ask $T2$ to release the resource and let $T1$ move on that;
- inserting the configuration parameters regarding the "time outs" into the database, in order to allow the user specify how long a Nodo or a Treno agent must wait for an answer.

Acknowledgments

This paper is based on Chapter 5 of Daniela Briola's Ph.D. Thesis, [2]. We thank Riccardo Caccia from Ansaldo STS for his help and support during all the stages of FYPA development.

References

[1] M. J. Berryman and S. D. Angus. Tutorials on agent-based modelling with NetLogo and network analysis with Pajek. In R. L. Dewar and F. Detering, editors, *Proceedings of the 22nd Canberra International Physics Summer School*, pages 351–375, 2010.

[2] D. Briola. *Negotiation in Multiagent Systems: Protocols, Ontologies and Applications*. PhD thesis, DISI, University of Genova, Italy, 2011.

[3] D. Briola and V. Mascardi. Multi agent resource allocation: a comparison of five negotiation protocols. *In this volume*.

[4] D. Briola, V. Mascardi, and M. Martelli. Intelligent agents that monitor, diagnose and solve problems: Two success stories of industry-university collaboration. In *Journal of Information Assurance and Security*, volume 4, pages 106–117, 2009.

[5] D. Briola, V. Mascardi, M. Martelli, R. Caccia, and C. Milani. Dynamic resource allocation in a MAS: A case study from the industry. In *From Objects to Agents Workshop, WOA 2009, Proceedings*, 2009.

[6] Uri Wilensky, at the Center for Connected Learning (CCL) and Computer-Based Modeling, Northwestern University, Evanston, IL. *Netlogo: Reference homepage*, 1999. http://ccl.northwestern.edu/netlogo/.

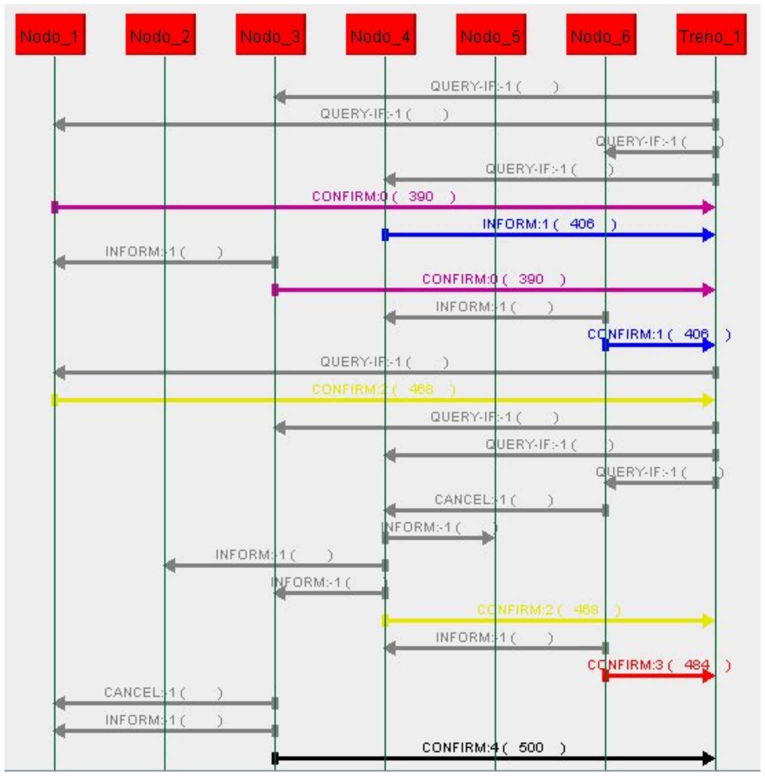

Figure 9. Example 3: JADE sniffer view

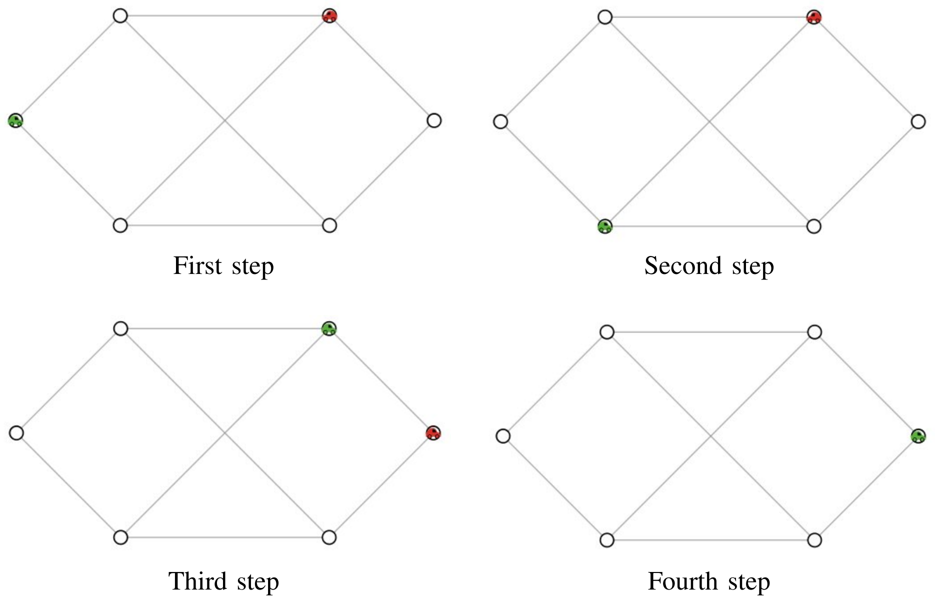

Figure 10. Example 3: NetLogo screenshots

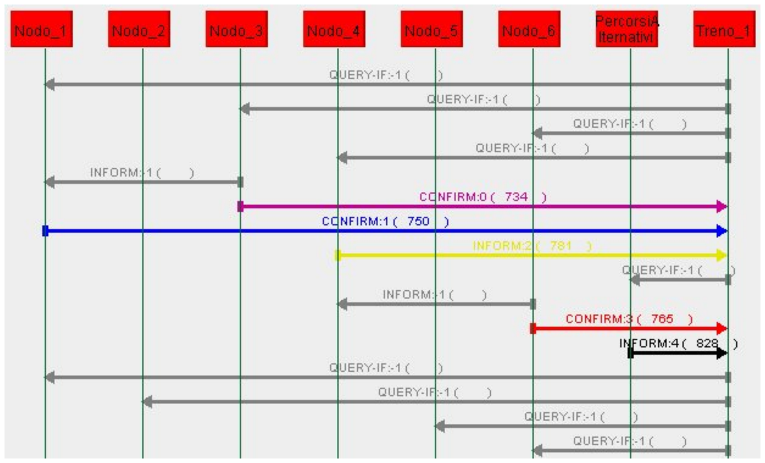

Figure 11. Example 4: JADE sniffer view

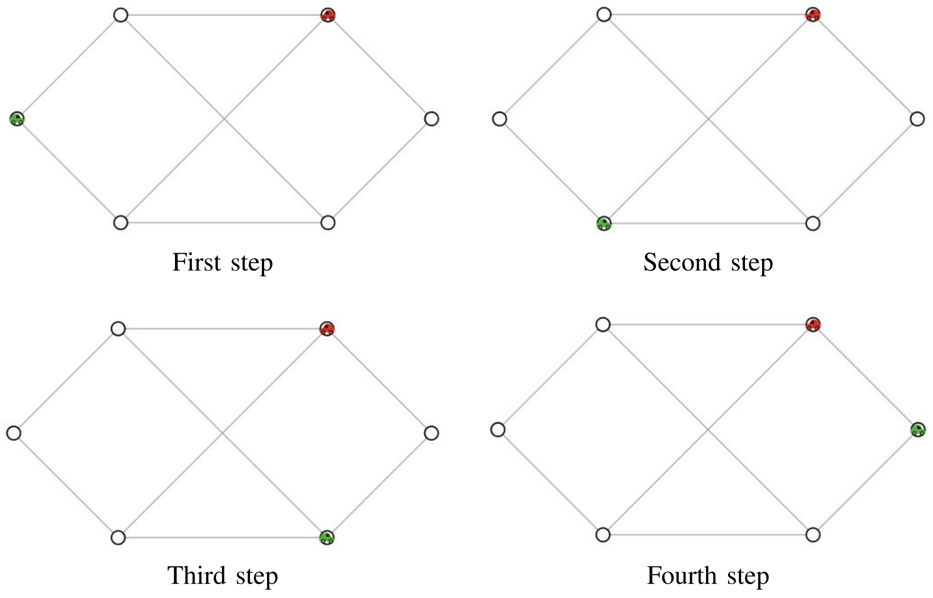

Figure 12. Example 4: NetLogo screenshots