The Architecture of an Emergency Plan Deployment System

MARCELO TÍLIO MONTEIRO DE CARVALHO [1]
JULIANA FREIRE[1]
MARCO ANTONIO CASANOVA[2]
[1] {tilio, jufreire}@tecgraf.puc-rio.br
Grupo de Tecnologia em Computação Gráfica / TeCGraf
[2] casanova@inf.puc-rio.br
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 - Rio de Janeiro, RJ - Brasil - CEP 22453-900

Abstract. This paper outlines the architecture and early implementation of an emergency plan deployment system that helps teams of human agents develop and execute comprehensive emergency plans, hyperlinked to conventional as well as geographical documents. The architecture features six components: a document database; a plan management module; a resource management module; a geographical document management module; a conventional document management module; and the plan monitoring module. Central to the architecture is a simple plan modeling language that helps overcome some of the limitations of conventional emergency plan description schemes.

1. Introduction

Emergency plans are often expressed as lists of conditional tasks, some of which may refer to ancillary documentation, such as maps, simulation results, lists of authorities to contact, etc. The data is traditionally compiled into lengthy guides that teams of human agents access in response to emergency situations.

Information systems designed to help deploy emergency plans must achieve two important design goals. First, they must help monitor the execution of emergency plans, which are carried out by teams of human agents. Second, they must help organize and access the often-vast collection of ancillary documentation.

This paper introduces the architecture of an emergency plan deployment system that achieves these two design goals, with improved functionality. The system features a fully distributed architecture where mobile devices communicate with fixed sites to: (1) monitor plan execution; (2) search, retrieve and visualize complex documents. The system permits designing emergency plans with a complexity similar to that of PERT diagrams [MPD83]. Yet, the human agents, working in the field, will be able to conveniently follow the plan, with the help of the mobile devices. The system therefore mimics the cooperative, distributed multi-agent environment typical of emergency response teamwork.

The implementation of the system will follow a two-step strategy. The first stage, currently under development, offers a flexible plan modeling language and implements plan monitoring in a centralized environment. The second stage will cover the migration of the system to a distributed platform that incorporates mobile computing devices.

The ideas described in the paper represent an evolution of the InfoPAE System [InfoPAE], designed to manage and monitor emergency tasks covering incidents in oil and gas pipelines. The InfoPAE system was implemented by TecGraf for PETROBRÁS.

FRIEND [BTR94], INCA [IG99] and MokSAF [LPHLS99] are examples of emergency management systems. In particular, the MokSAF system supports route planning by combining AI techniques with GIS. Other examples of multi-agent systems with internal planning components are RETSINA [PKPSS99] and HIPaP [PSS00]. A survey of cooperative multi-agent systems appears in [Le99] and [FC99] provides an interesting application of plan generation in the context of databases.

The paper is organized as follows. Section 2 outlines the architecture of the system. Section 3 introduces the plan modeling language. Section 4 describes the current implementation. Section 5 discusses possible extensions to the language. Finally, section 6 contains the conclusions.

2. Outline of the architecture

The emergency plan deployment system has six basic components:

- the document database
- the plan management module
- the resource management module
- the geographical document management module
- the conventional document management module
- the plan monitoring module

The database persistently stores documents that represent plans, plan frameworks, plan states, resources, and geographical and conventional data. In addition to predefined relationships, the database permits hyperlinking documents as a flexible way to model binary document relationships.

From the point of view of the database, a plan is just a complex document recursively built upon other documents that describe elementary plans, that is, questions about the current scenario and tasks that must be performed. As such, a plan, or one of its components, may be hyperlinked to other documents in the database.

A plan framework, as the name indicates, is a framework that describes a generic plan scheme that can be instantiated to generate new plans.

A plan state describes the current state of a plan execution and is an internal object of the plan monitoring module.

A resource is any equipment, facility or human agent that emergency plans need. Typically, resources have a rich categorization that the database must reflect. For example, sea fences, used to contain oil spills, have more than 10 categories.

The database also stores geographical and conventional data, such as facility maps, simulation results, lists of authorities to contact, etc. These documents provide valuable information to the deployment of emergency plans and must be readily available to the human teams.

The plan management module incorporates four tools to help users create and maintain plans:

- a plan browser to search for plans in the database
- a task editor to create new tasks and relate them to resources and other documents in the database
- a plan editor to compose more complex plans, using the plan modeling language or plan frameworks
- a plan framework editor to create, store and reuse plan frameworks

The resource, the geographical and the conventional document management modules offer database interfaces to retrieve, insert and update these types of documents and need no further comment at this point.

Finally, the plan monitoring module is responsible for plan execution. It consists of:

- the plan monitoring engine
- the field user module
- the central coordination module

The plan monitoring engine, as the name implies, monitors the execution of the plan by human agents. It persistently maintains the current plan state in the database, interacting with the human agents to register the progress of tasks contained in the plan (see Section 3.2).

The field user module helps human agents keep track of the current state of the plan assigned to them. It also helps them search and retrieve ancillary information required to execute tasks, locate resources, etc.

The central coordination module is designed to help the upper management staff keep track of the current scenario: the current assessment of the situation (as reported by the human agents), what has been accomplished, what remains to be done, etc.

The final implementation will feature a distributed architecture where mobile devices communicate with fixed sites to monitor plan execution. This implies that the mobile devices will run the field user interface and will contain, depending on their capacity, replicas of the documents required by the tasks.

A mobile device may also run specialized versions of the resource, the geographical and the conventional document management modules to help search and visualize such documents. This opens an array of possibilities, including voice output synthesis and other less conventional data rendering techniques.

Section 4 details the current stage of the implementation, which already covers sophisticated versions of the plan editor and of the plan monitoring module.

3. The plan modeling language

Section 3.1 introduces the syntax while Section 3.2 outlines the semantics of the language. Section 3.1 also intuitively explains the constructs of the language.

3.1 Syntax

The syntax of the language, in BNF notation, is shown in Figure 1.

An example of a toy plan P is:

(1) $P = (Q_1;A_1 \setminus Q_2;A_2) \; ; \; (B_1 \; // \; B_2)$

where

Q_1 = "Is the oil that spilled heavy?"

A_1 = "Stop pump P_1 and close valve V_1"

Q_2 = "Is the oil that spilled light?"

A_2 = "Stop pump P_2 and close valve V_2"

B_1 = "Call the Port Authority"

B_2 = "Call the Facility Manager"

Intuitively, a plan is recursively built out of tasks using sequential, parallel and alternative compositions.

Although not distinguished in Figure 1, there are three types of tasks - *undeferrable activities, deferrable activities* and *tests*. Depending on its type, the execution of a task goes through several states, as described in Section 3.2, that the human agent in part controls by sending messages to the system.

An *undeferrable activity* describes an activity a human agent must perform. He informs the system when he starts and when he finishes the activity, by sending *start* and *finish*.

A *deferrable activity* also permits the human agent to defer it for later execution, by sending *defer,* even after he started the activity. This liberality impacts the semantics of plans, but it reflects real-world plan modeling requirements.

A *test* urges the human agent to check for some the real-world condition. He can either validate the condition, by sending *finish*, or he can reject it, by sending *reject*.

The *sequential composition* $(P_1;P_2)$ says that the human agent must execute P_1 before P_2.

The *parallel composition* $(P_1//P_2)$ defines that the human agent must execute P_1 and P_2 in parallel.

The *alternative composition* $(P_1\setminus P_2)$ indicates that the human agent can execute one of the plans and reject the other, or that he can execute both.

```
<plan>   ::= <sequential plans>  |
             <parallel plans>    |
             <alternative plans> |
             <task>
<sequential plans>
         ::= '('<plan>';'<plan>')'
<parallel plans>
         ::= '('<plan>'//'<plan>')'
<alternative plans>
         ::= '('<plan>'\'<plan>')'
<task>   ::= <string>
<string> ::= any string not containing one
             of the reserved symbols:
             ";", "//", "\", "(", ")"
```

Figure 1: Syntax of the plan modeling language

We now define additional concepts that will be used in Section 3.2.

P is the *father* of P_1 and P_2, and P_1 and P_2 are *children* of P, iff P is of the form $(P_1;P_2)$, $(P_1//P_2)$ or $(P_1\setminus P_2)$.

Q is a *component* of P iff Q is a child of P or Q is a component of a component of P.

P_1 is a *pre-requisite* of P_2 in plan P iff P has a component of the form $(P_1;P_2)$.

The parallel and the alternative compositions are commutative and all three compositions are associative. Hence, as an abuse of syntax, we admit expressions of the form $(P_1;...;P_n)$, $(P_1//...//P_n)$ or $(P_1\setminus...\setminus P_n)$.

Tests and the alternative composition are powerful tools to build plans. However, a less experienced plan developer may create ill-formed plans that have tests outside alternative compositions thereby causing plan components to be rejected without offering alternatives.

For example, if our toy plan P were:

(1') $P' = A_1 \; ; \; Q_1 \; ; \; (B_1 \; // \; B_2)$

then it would be ill-formed. We do not consider this plan acceptable for the following reason. Suppose that the human agent performs the following actions:

- executes A_1 ;
- informs that Q_1 is false and stops following P'.

Hence, P' will fail after being partly executed, leaving the effects of A_1 visible (i.e., "pump P_1 stopped and valve V_1 closed"). By contrast, a well-formed plan would offer an alternative to Q_1 , say, to undo the effects of A_1 (i.e., "restart pump P_1 and re-open valve V_1") before abandoning P'.

For this reason, we introduce two new concepts. A *guarded plan* is a plan of the form $(T_1;P_1\backslash...\backslash T_n;P_n)$, where T_i is a test and P_i is a plan, for each $i\in[1,n]$. A plan P is *well-formed* iff the only occurrences of tests and the alternative composition in P are in components of P that are guarded plans.

Finally, we define a *plan framework* as a plan, except that syntactical variables can be used to replace sub-plans. For example, we may alter (1) to become the plan framework in (8):

(8) $(<t_1>;<a_1>\backslash<t_2>;<a_2>);(<b_1>//<b_2>)$

where the symbols enclosed in angular brackets are syntactical variables ranging over plans.

Intuitively the plan designer will use a plan framework to create new plans by instantiating the syntactical variables with plans, possibly selected from the database.

3.2 Semantics

The semantics of the language clarifies what is a plan execution. We opted for a semantics that maps a plan into a set of automata that interact with each other.

The set of automata associated with a plan is inductively defined in a way that mimics the language constructs.

In Figure 2, labels in italics represent state transitions the human agent controls, while labels in boldface represent those resulting from the interaction among the automata.

The automata for undeferrable activities, deferrable activities and tests are shown in Figures 2a, 2b and 2c, respectively.

The automaton that governs sequential, parallel and alternative compositions is again that in Figure 2c, except that all labels are in boldface.

Finally, the automaton for the plan itself is shown in Figure 2d.

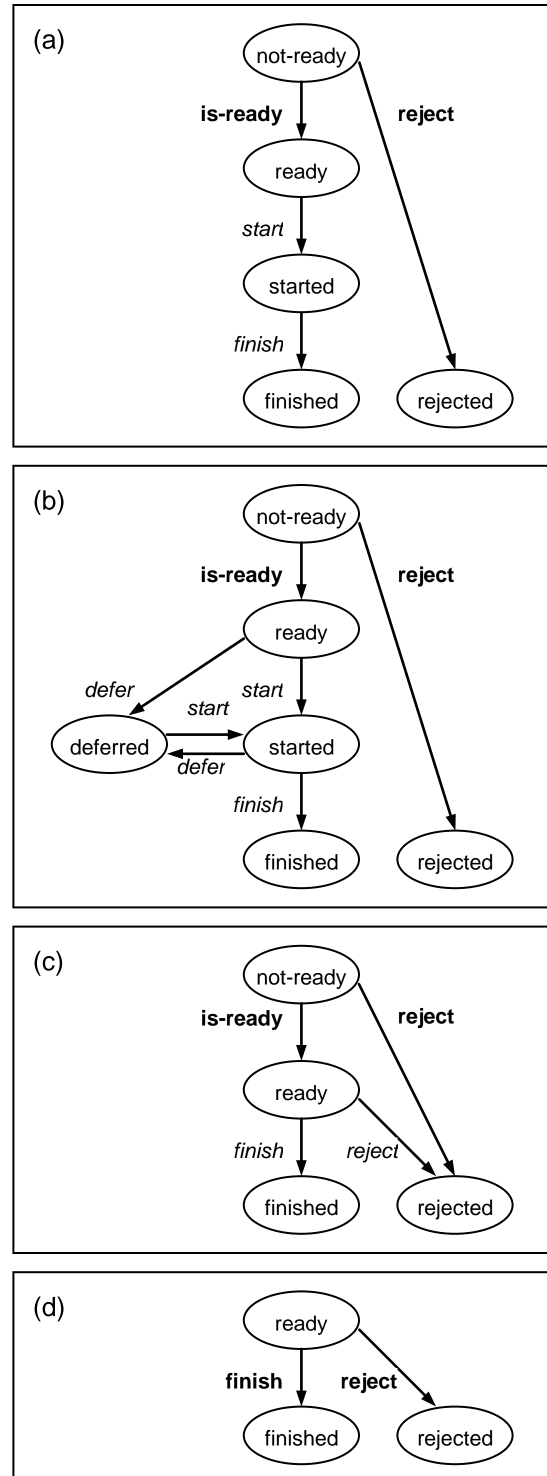


Figure 2: Component Automata.

We say that a component Q *reaches* state S iff the automaton associated with Q changes to state S. The state transitions are concisely defined as follows:

Any type of component (Figures 2a,b,c):

not-ready to ready -

the father of the component reaches *ready;* or

the pre-requisite of the component reaches *finished* or *deferred.*

not-ready to rejected - the pre-requisite or the father of the component reaches *rejected.*

Undeferrable activity (Figure 2a):

ready to started - human agent sends *start.*
started to finished - human agent sends *finish.*

Deferrable activity (Figure 2b):

ready to started - human agent sends *start.*
ready to deferred - human agent sends *defer.*
deferred to started - human agent sends *start.*
started to deferred - human agent sends *defer.*
started to finished - human agent sends *finish.*

Test (Figure 2c):

ready to finished - human agent sends *finish.*
ready to rejected - human agent sends *reject.*

Sequential composition $(P_1;P_2)$ ***(Figure 2c):***

ready to finished -
P_i reached *finished* or *deferred,* for $i\in[1,2]$.
ready to rejected - P_1 or P_2 reached *rejected.*

Parallel composition $(P_1//P_2)$ ***(Figure 2c):***

ready to finished -
P_i reached *finished* or *deferred,* for $i\in[1,2]$.
ready to rejected - P_1 or P_2 reached *rejected.*

Alternative composition $(P_1\backslash P_2)$ ***(Figure 2c):***

ready to finished -
P_i reached *finished* or *deferred,* for $i\in[1,2]$*; or*
P_1 reached *finished* or *deferred* and P_2 *rejected; or*
P_2 reached *finished* or *deferred* and P_1 *rejected.*

ready to rejected - P_1 and P_2 reached *rejected.*

Plan (Figure 2d):

ready to finished - the composition, or the task, that makes up the plan reached *finished* and no component of the plan is *deferred.*

ready to rejected - the composition, or the task, that makes up the plan reached *rejected.*

This concludes the description of the semantics.

4. Current implementation

This section summarizes the current implementation of the plan management and the plan monitoring modules. The annex contains a sample screen of the current implementation.

4.1 The Plan Management Module

The current implementation of the plan management module offers a version of the plan modeling language that covers only well-formed plans, with some syntactic sugaring.

The implementation adopts function calls as the basic mechanism to express the language constructs.

A task A is expressed as a function

(9) `function A()`

A guarded plan $(W_1;A_1\backslash...\backslash W_n;A_n)$ is expressed with the help of the `Ask` function:

(10) $\texttt{Ask}(Q,\{W_1,...,W_n\},\{A_1,...,A_n\})$

where

Q is any character string, interpreted as a question to be posed to the human agent

$W_1,...,W_n$ is a list of tests that represent acceptable answers to question Q.

$A_1,...,A_n$ is a list of alternative plans

The human agent may select one or more tests. If he selects W_i, then he is signaling to change the state of W_i from *ready* to *finished* and, consequently, to select plan A_i for execution. Hence, we treat Q as syntactic sugar that can be incorporated into the tests $W_1,...,W_n$.

The parallel composition $(A_1//...//A_n)$ is expressed with the help of the `Do` function:

(11) $\texttt{Do}(A_1,...,A_n)$

Likewise, the sequential composition $(A_1;...;A_n)$ is expressed by a sequence of calls to the `Do` function:

(12) $\texttt{Do}(A_1)$
...
$\texttt{Do}(A_n)$

The toy plan P in (1) is then expressed as:

(13)
```
function P()
  Do (P1)
  Do (P2)
end
function P1()
  Ask ("What oil spilled?",
       {"Heavy oil","Light oil"},
       {A1 , A2})
end
function P2()
  Do (B1, B2)
end
```

where

A_1 = "Stop pump P_1 and close valve V_1"

A_2 = "Stop pump P_2 and close valve V_2"

B_1 = "Call the Port Authority"

B_2 = "Call the Facility Manager"

The body of the function that expresses task A may also include the following function calls:

`Show("S")`, where S is a string.

Associates string S with task A. When task A reaches *ready,* string S is shown to the human agent. Exactly one call to `show` must occur in the body of `function A()`.

`AddType("T")`, where T is a string.

Associates a type T with task A. There can be none, one or many calls to `addtype` in the body of `function A()`.

`AddDoc("D")`, where D is a file name.

Associates a document, contained in D, with task A.

`AddInfo("C","T")`, where C and T are strings.

Associates textual information T with task A and classifies T in category C (an arbitrary string).

`OnStart(F)`, where F is a function call.

Associates a function F to be called when task A reaches *started*.

`Deferrable`

Indicates that the task is deferrable.

The string that `Show` associates with the task should be a short sentence with concise instructions to the human agent. More extensive information should be passed using `AddDoc` or `AddInfo`. The current implementation extends `AddInfo` to retrieve information from the database, instead of merely passing textual information as a parameter.

Note that the types created with the help of `AddType` are not related to the three task types introduced in Section 3.1. They simply let the user create his classification for tasks. The current implementation uses task types, defined with the help of `AddType`, to filter the tasks the human agent has access.

A full example of a task definition is:

(14)
```
function A()
  Show    ("Stop pump P1 and
            close valve V1")
  AddDoc  ("pump-manual.doc")
  AddDoc  ("valve-manual.doc")
  AddType ("shutdown")
  AddInfo ("phone", "2222222")
  AddInfo ("e-mail",
           "person@place.com")
end
```

The string "Stop pump P_1 and close valve V_1" will then be shown when the task reaches *ready*. The files "pump-manual.doc" and "valve-manual.doc" contain reference manuals the human agent can access. The task also has two categories of associated textual information, "phone" and "e-mail", that indicate, say, the contact phone number and the e-mail of people that must be notified when task is executed.

4.2 The Plan Monitoring Module

The plan monitoring module implements the abstract machine outlined in Section 3.2. It emulates the execution of a plan P by interpreting the statements of `function P()`. The monitor also uses an auxiliary structure that keeps the state of all tasks in the plan.

For example, consider plan P in (13). The monitor starts by moving P_1 to the *ready* state. The human agent is then posed with the question "What oil spilled?". If he selects "Heavy oil", then the state of this test is changed to *finished* and, consequently, the state of A_1 is changed to *ready*. If, on the other hand, he selects "Light oil", then the state of this second test is changed to *finished* and the state of A_2 is changed to *ready*.

After the state of A_1 (or of A_2) changes to *finished*, the states of B_1 and B_2 will both change to *ready.* The monitor will then select both B_1 and B_2, prompting the human agent to "Call the Port Authority" and to "Call the Facility Manager". The human agent may then execute these instructions in any order.

5. Extensions

We briefly discuss in this section extensions to the language and to the plan monitoring module.

A task has an implicit duration, which is the time interval the human agent takes to perform it. Now, observe that the sequential and the parallel compositions reflect just two temporal relationships between tasks. For example, we may relate two tasks by forcing them not to overlap, or by forcing them to finish at the same time, etc. Continuing with this line of reasoning, we may redesign the plan modeling language to incorporate an algebra of tasks that mimics a time interval algebra [BMN00], or to follow the style of multimedia presentation languages, such as SMIL [W3C], that specifies the presentation of several media objects in time-space.

Plan monitoring can also be extended to account for backtracking and dynamic plan relaxation. Indeed, the `Ask` function can be relaxed to allow the human agent to go back to a question (a call to `Ask`) and change his answer, perhaps to reflect new conditions that developed during the execution of the plan. This extension requires changing the semantics of the language to account for backtracking of plan execution. It may also require introducing the concept of *compensatory tasks* to undo real world activities previously carried out by the human agent.

6. Conclusions

This paper outlined the architecture of an emergency plan deployment system that helps human teams develop and execute comprehensive emergency plans, hyperlinked to conventional as well as geographical documents.

The current implementation permits experimenting with the language to model realistic plans, which we are currently doing in the context of the InfoPAE Project.

Acknowledgements

We wish to thank Antonio Furtado and Angelo Ciarlini, from PUC-Rio, for their careful reading of an earlier version of this paper and Angelo Francisco dos Santos, from PETROBRÁS, for his many contributions to the ideas expressed in this paper.

References

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. "Temporal Logics for Real-Time System Specification", *ACM Computer Surveys,* Vol. 32, No. 1 (March 2000), 12--42.

[BTR94] B. Bruegge, K. O'Toole and D. Rothenberger, "Design Considerations for an Accident Management System". In *Proc. of the Conference on Cooperative Information Systems* (1994), 90--100.

[FC99] A. Furtado and A. Ciarlini. "Operational Characterization of Genre in Literary and Real-life Domains". In *Proc. of the ER'99 Conceptual Modelling Conference,* Paris, France (1999).

[IG99] W. Iba, M. Gervasio. "Adapting to User Preferences in Crisis Response". In *Proc of Intelligent User Interfaces* (1999), 87--90.

[InfoPAE] Sistema de Informação para Apoio a Planos de Ação de Emergência. Manual do Usuário. Version 1.1. TeCGraf, PUC-Rio (2001).

[Le99] V. Lesser, "Cooperative Multiagent Systems: A Personal View of the State of the Art", *Knowledge and Data Engineering*, Vol. 11, No. 1 (1999), 133--142.

[LPHLS99] T. Lenox, T. Payne, S. Hahn, M. Lewis, and K. Sycara, "MokSAF: How should we support teamwork in human-agent teams?". Tech. Report CMU-RI-TR-99-32, Robotics Institute, CMU (1999).

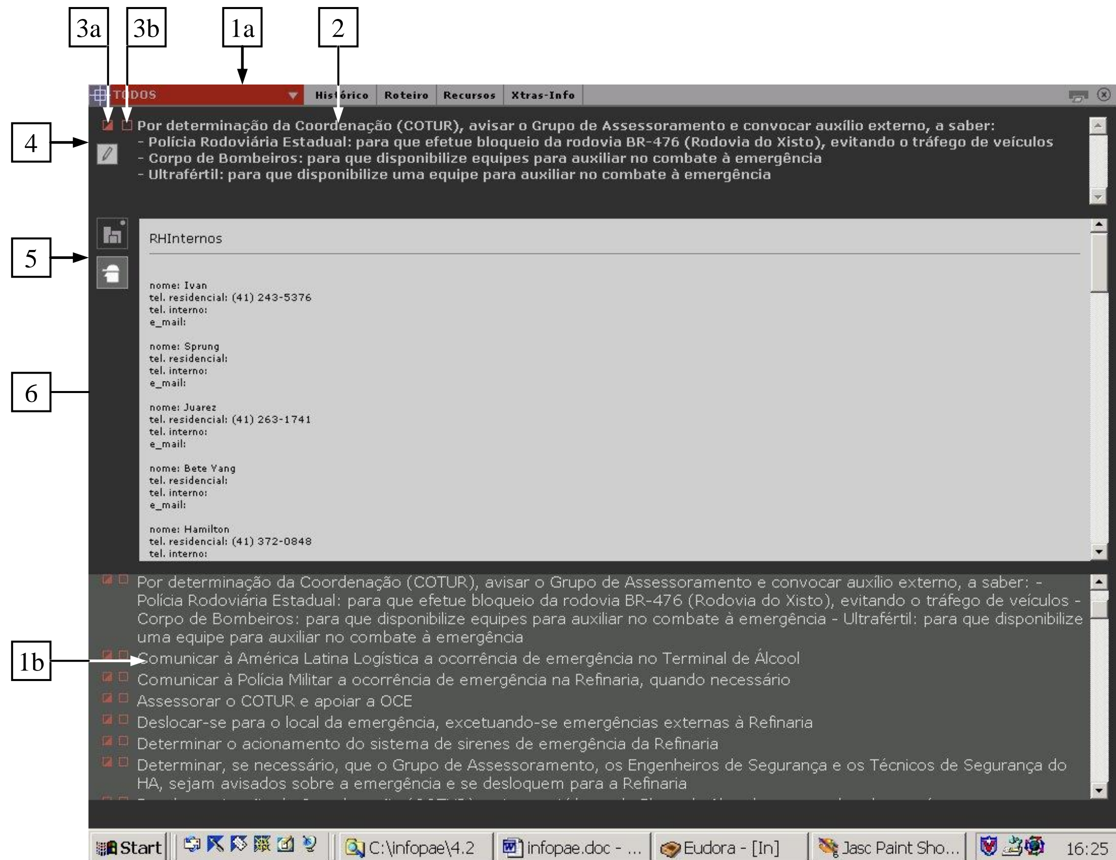
[MPD83] J. J. Moder, C. R. Phillips and E. W. Davis, Project Management with CPM, PERT and Precedence Diagramming (3rd. ed.), *Van Nostrand Reinhold*, New York (1983).

[PKPSS99] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara. A Planning Component for RETSINA Agents. In *Agent Theories, Architectures, and Languages* (1999), 147--161.

[PSS00] M. Paolucci, O. Shehory, and K. Sycara. "Interleaving planning and execution in a multiagent teamplanning environment." Tech. Report CMU-RI-TR-00-01, Robotics Institute, CMU (2000).

[W3C] World Wide Web Consortium (W3C), *SMIL 2.0.* http://www.w3.org/AudioVideo/ (last visited on June 18th, 2001).

Annex - Sample screen from the current implementation



1. The user may select, from the set of tasks that are ready, the subset of all tasks of a given type:
   a. Pulldown menu used to choose the desired task type.
   b. Panel showing the list of selected tasks.
2. The user may pick up a task from those selected, which will be shown on the top panel.
3. The user clicks on:
   a. the button on the left to inform that he started the task, and
   b. the button on the right to indicate that he finished the task.
4. The user clicks on this button to add comments to the task in a new window that will pop up.
5. The interface displays a list of predefined icons representing the information categories associated with the task (via `AddInfo`).
6. Panel showing any additional information associated with the task (via `AddInfo`), classified by category.