Journal of the Brazilian Computer Society, 2009; 15(4):41-55. ISSN 0104-6500

Aspects of planning support for human-agent coalitions

Clauirton de Albuquerque Siebra[1,2]*, Natasha Correia Queiroz Lino[1,2]

[1]Department of Informatics, Federal University of Paraiba, Cidade Universitária – CCEN, 58059-900, João Pessoa, PB, Brazil

[2]Centre for Intelligent Systems and their Applications, The University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, UK

Received: September 12, 2009; Accepted: December 22, 2009

Abstract: This paper analyses aspects associated with the development of joint human-agent planning agents, showing that they can be implemented, in a unified way, via a constraint-based ontology and related functions. The constraints' properties have already been used by several planning approaches as an option to improve their efficiency and expressiveness. This work demonstrates that such properties can also be employed to implement collaborative concepts, which are maintained transparent to the planning mechanisms. Furthermore, the use of constraints provides several facilities to the implementation of advanced mechanisms associated with the human interaction, as also demonstrated here.

Keywords*: multiagent planning, collaboration, constraint-based ontology, human-agent interaction.*

1. Introduction

Coalition, from Latin *coalescere* (*co-*, together + *alescere*, to grow) is a type of organisation where joint members work together to solve mutual goals. The principal feature of a coalition is the existence of a global goal, which motivates the activities of all coalition members. However, normally such members are not directly involved in the resolution of this goal, but in sub-tasks associated with it. For example, in a search and rescue coalition that aims to evacuate an island (evacuate island is the global goal), there are several sub-tasks (e.g., allocate helicopters, provide information about road conditions, etc.) that must be performed to reach the global goal. Members of a coalition commonly need to collaborate because they have limits in abilities and knowledge. For example, ambulances require a clear route to rescue injured civilians. If the route is blocked, ambulances must ask for help from truck units so that they unblock the route.

Considering the diversity of tasks of a coalition, it is natural that its components carry out different planning and plan execution activities at different decision levels. In fact, one of the principal aims of knowledge based tools for coalitions is to integrate and support such components so that they are able to work synergically together, each of them accounting for part of the planning and execution process.

The use of planning assistant agents[6,25] is an appropriate option to provide this kind of support and integration. Agents can extend the human abilities and be customised for different planning activities performed along hierarchical decision-making levels. However, the use of standard planning mechanisms is not sufficient to deal with the complexity of problems associated with coalition domains. In these domains, activities cannot consist merely of simultaneous and coordinated individual actions, but they must also be developed on a collaborative framework that ensures an effective mutual support among joint members[8]. Furthermore, there are also aspects related to the human participation, in conjunction with (semi) autonomous agents, that must also be considered and improved during the development of a coalition support application.

This paper is concerned with the specification of a framework to the development of hierarchical coalition support systems. Our core idea is to consider requirements associated with the implementation of multiagent planning mechanisms[21] because they can bring several advantages to coalition operations such as prediction of failures, resource allocation, conflict identification/resolution and so on. However this framework must also be able to support other important requirements for coalition systems, related to collaborative theorie[4,8,9,12] and human-agent interaction[6,15,25]. The problem is that the requirements that we intend to consider are investigated by different research areas rather than in a unified way. Consequently the solutions are generally incompatibles or hard to integrate and there is a lack in the current literature about frameworks that discuss such issue.

In this context, the main aim of this work is to show how several design aspects of coalition support systems can be specified, in a unified way, via a constraint-based ontology and related functions[18]. The advantage of this approach is that the implementation of all requirements can be understood from a unique perspective and implemented on the same basis, using constraints representation and manipulation. The contribution associated with this framework specification can be divided into two parts. First, the extension and adaptation of a constraint-based ontology (<I-N-C-A>[26]). Second, the definition of a set of functions to manipulate the elements of this ontology. We consider that all agents of a coalition will be based on the ideas discussed here, making commitments

*e-mail: clauirton@di.ufpb.br; c.siebra@ed.ac.uk

on the same ontology and concepts. Thus, we do not intend to integrate pre-defined coalition support systems such as proposed in other works[2,29].

The remainder of this document is organised as follows: Section 2 defines the problem that this work intend to tackle and related domain. Section 3 presents an agent-centred view of the planning process, classifying different sets of constraints and functions that are related to this process. Section 4 summarises the <I-N-C-A> representation, a constraint-based general purpose ontology that was used to specify the constraint model. Section 5 discusses the synthesis of several coalition systems aspects, which are conceptually classified into three groups: planning agent aspects, teamwork aspects and human interaction aspects. Section 6 presents some details about the implementation approach, which is mainly based on the ideas of *constraint managers* and *activity handlers*. Section 7 discusses the main related works, pointing out the differences regarding our proposal. Finally, Section 8 concludes with a critical vision about this work, stressing the opportunities for improvements.

2. Problem Definition

Imagine a disaster relief operation where rescue units, such as fire brigades, are planning and executing tasks. Each of these rescue units has an assistent agent that is delivering planning information to and receiving information from its human user. At this moment we do not need to consider the kind of device where the computational agent is running, which could be a handset-like device (tactical level), a laptop (operational level) or a mainframe (strategic level). The set of human-agent components makes up a multiagent environment, which is organised as a hierarchical structure.

We consider three decision-making levels in this hierarchy: strategic, operational and tactical. Higher level agents decompose complex tasks and delegate its parts to their subordinate agents in lower levels. In this way, we do not have a flat organisation once levels of decision have different responsibilities and knowledge, similar to a military organisation. We also consider that all coalition has a global goal, which is also a mutual goal. This means, all members are direct or indirectly involved in reaching this goal. We do not have competition, neither malicious agents. All decisions must be carried out in favor of the coalition rather than individuals. All these features characterise and restrict the kind of domain that we intend to tackle.

As we have a hierarchical multiagent planning, where each agent is an assistant agent, we must define an approach to the planning process. A natural example could be the *Distributed Hierarchical Task Network* (DHTN) approach[30]. However, the focus of DHTN and other distributed planning approaches is on typical planning issues such as task decomposition, replanning, plan communication and conflict detection/resolution. This fact raises design problems when we try to extend such planning systems with collaborative concepts. The principal reason, as discussed in[8], is that *collaboration between different problem-solving components must be designed into systems from the start. It cannot be patched on*. The problem here is how to incorporate collaborative requirements into a distributed planning process, so that the final joint plan is not just a sum of individual plans.

Considering that one of the most important functions of knowledge-based tools for coalitions is to support human users, and this is the main reason to use assistant agents, we also need to understand how human users interact in the collaborative planning process. Note that we are not considering aspects of interface that could improve the human-agent interaction[17]. Our focus is on the study of the additional requirements that the human presence brings to the development of collaborative planning agents. This important issue for coalition systems is not well explored by noteworthy collaboration theories, mainly because such theories only consider agent-agent interactions rather than human-agent interactions[23]. These issues are discusses in details in the next sections.

3. A Constraint-Based View

An appropriate way to understand the planning process of assistant agents is via the elements that have influence on its reasoning (e.g., world state, time, human users, etc.). Considering that each of such elements is represented as a set of constraints (*Ci*), we have the architecture illustrated in the next figure (Figure 1). According to this architecture, we can identify the following sets of constraints:

- C_0: represents the set of constraints imposed by the environment such as weather conditions or the state of world objects;
- C_1: represents the set of temporal constraints that restricts the planning process in adding new activities into the agent's agenda (plan);
- C_2: represents the capabilities and internal state of agents that, together, restrict the kind and amount of activities that they can perform. Note that the set C_2, from subordinate agents, also restrict the creation of plans by its superior;
- C_3: represents the constraints associated with the process of commitment. Each delegated activity has one constraint C_3, whose value indicates if a specific agent is committed to the performance of such an activity. The value of C_3 is generated via the commitment/report function F_1;
- C_4: represents the set of constraints used by humans to control/customise the behaviour of agents;
- C_5: represents the set of constraints that restricts the options of a human user;
- C_6: represents the set of constraints associated with activities of other agents. The function F_2 acts on C_6 to discover possible inconsistencies in such a set, so that agents can mutually support each other;
- C_7: represents the set of all constraints monitored by the function F_3, which accounts for generating explanations to human users;

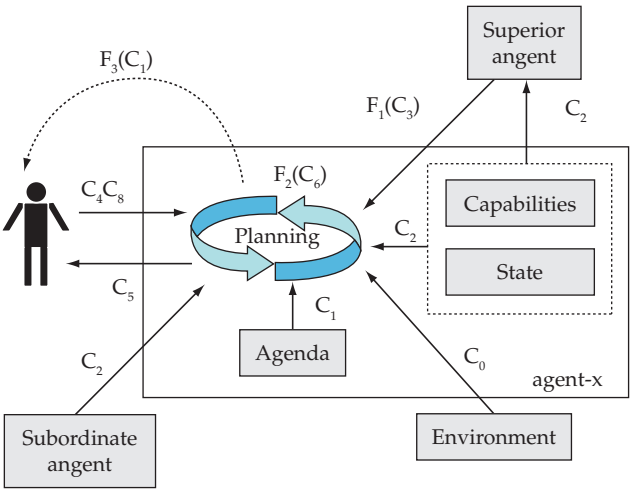

Figure 1. Planning architecture of an assistant agent.

```
PLAN ::=
  <plan>
     <plan-variable-declarations>
       <list> PLAN-VARIABLE-DECLARATION </list>
     </plan-variable-declarations>
     <plan-issues>
       <list> PLAN-ISSUE </list>
     </plan-issues>
     <plan-issue-refinements>
       <list> PLAN-ISSUE </list>
     </plan-issue-refinements>
     <plan-nodes>
       <list> PLAN-NODE </list>
     </plan-nodes>
     <plan-node-refinements>
       <list> PLAN-NODE-refinement </list>
     </plan-node-refinements>
     <constraints><list>CONSTRAINT</list></constraints>
     <annotations><map>MAP-ENTRY</map></annotations>
  </plan>
```

Figure 2. First level of the <I-N-C-A> schema to plans.

- C_8: represents the set of constraints that accounts for providing a declarative manner of restricting the autonomy of the planning process.

Two simple properties can be defined to the constraint sets C_0...C_8. First, while some sets are composed of only one type of constraint (e.g., C_0: world-state, C_1: time and C_2: resource), other sets (e.g., C_6 and C_7) can be composed of several constraint types defined in the model. Second, a constraint c can appear in one or more sets, so that a constraint may not uniquely be associated with a specific set.

4. <I-N-C-A> Ontology

<I-N-C-A> (Issues - Nodes - Constraints - Annotations) is a general-purpose ontology that can be used to represent plans (Figure 2) as a set of constraints on the space of all possible behaviours in an application domain[26]. Planning can be described as synthesizing an <I-N-C-A> object, i.e., a plan, in which nodes are activities. We can formally define an <I-N-C-A> object as a 4-tuple (<I,N,C,A>) consisting of: a set of issues I, a set of activity nodes N, a set of constraints C and a set of annotations A. All these elements can be exchanged among agents of a coalition, so that <I-N-C-A> also accounts for formatting the content of the messages protocol. This should already be expected, once <I-N-C-A> is an ontology and one of the main aims of a ontology is to enable the sharing of knowledge[10]. The next subsections detail the formalism[28] behind the <I-N-C-A> components and the relation among them.

4.1. Issues (I)

I is the set of unresolved issues in the current plan, i.e., in this <I-N-C-A> object. An issue is represented by a syntactic expression of the form l:M(O_1,…,O_n), where:

- l is a unique label for this issue;
- M is a symbol denoting a primitive plan modification activity;
- O_1,…,O_n are plan-space objects, i.e. they are issues, nodes, constraints or annotations. The number n of such objects, and the interpretation of each object in the context of the issue, will depend on the particular primitive plan modification activity represented by this issue.

Issues can be seen as primitive meta-level activities, i.e. things that need to be done to the plan before it becomes a solution to a given planning problem. The most commonly found primitive meta-level activities carried out by planners, but usually only implicit in their underlying implementation or internal plan representation, are:

- Achieving a goal (in classical planners): let p be a world-state proposition and τ be a time point. Then the primitive meta-level activity of achieving p at τ can be represented as the issue:

$$l_1\texttt{:achieve(p,}\tau\texttt{)}$$

- Accomplishing a complex activity (in HTN planners): Let a be a complex activity. Then the primitive meta-level activity of accomplishing α can be represented as the issue:

$$l_2\texttt{:refine(}\alpha\texttt{)}$$

Here, `achieve` and refine are examples of symbols denoting primitive plan modification activities. Note that these symbols are not domain specific but specific to the planning process by which these types of issue are handled. Issues can be either 'negative', in which case they can be thought of as flaws in the plan, or they can be 'positive', e.g., opportunities.

4.2. Nodes (N)

N is the set of activities (nodes) to be performed in the current plan, i.e., in this <I-N-C-A> object. An activity is a syntactic expression of the form l:α (o_1,…,o_n), here:

- l is a unique label for this activity,
- α is a symbol denoting an activity name;

- $o_1,\ldots,o_n$ are object-level terms, i.e. they are either constant symbols describing objects in the domain, or they are as yet uninstantiated variables standing for such objects.

In the context of <I-N-C-A>, nodes represent the object-level activities in the plan, i.e., things that need to be performed by some agent to execute the plan. Activities can be of two types from the perspective of the planner:

- Primitive activities: they can be carried out directly by an agent executing the plan. For example, in a search and rescue domain, the primitive activity of flying the aircraft ac1 from location loc1 to location loc2 may be represented as:

$$l_3\text{:fly(ac1,loc1,loc2)}$$

- Complex activities: they cannot be accomplished directly by the agent executing the plan but need to be refined into primitive activities. For example, the complex activity of rescuing an isolated person ip may be represented as:

$$l_4\text{:rescue(ip)}$$

In this example, fly is a primitive activity symbol and rescue is a complex activity symbol in some domain. Activity symbols have to be domain specific. It follows that there has to be an activity schema defined for the domain that has the name fly and describes when this activity schema is applicable and how it will change the world when applied, and there has to be a refinement defined in the domain that accomplishes a complex activity with the name rescue and describes how exactly it can be accomplished.

Note that the set N of activities in the plan may contain both complex activities and the primitive activities that have been chosen to implement them.

4.3. Constraints (C)

C is the set of constraints that must be satisfied by the current plan. A constraint is a syntactic expression of the form l:$c(v_1,\ldots,v_n)$, where:

- l is a unique label for this constraint,
- c is a symbol denoting a constraint relation, and
- $v_1,\ldots,v_n$ are constraint variables, i.e., they can represent domain objects (e.g., time points), variables in activities (which may have binding constraints attached).

Constraints represent the relations that must hold between the different objects related in the constraints for the plan to be executable. In this work we are interested in the use of the CONSTRAINT element (Figure 3) to define new types of constraints that represent the sets $C_0..C_8$, which were introduced in the previous section. According to <I-N-C-A> and considering a XML specification, a constraint is characterised by a type (e.g., world-state), a relation (e.g., condition or effect) and a sender-id attribute to indicate its source.

```
CONSTRAINT ::=
<constraint type="SYMBOL" relation="SYMBOL"
sender-id="ID">
      <parameters><list> PARAMETER </list></parameters>
      <annotations><map> MAP-ENTRY </map></annotations>
</constraint>
```

Figure 3. Specification of constraints.

The constraint content is described as a list of parameters, whose syntax depends on the type of the constraint. For example, a world-state constraint (*C0*), which is already a known-constraint in the <I-N-C-A> definition, has as parameter a list of PATTERN-ASSIGNMENT, which is defined as a pair pattern-value such as ((speed wind), 35 km/h). In XML specification we have:

```
CONSTRAINT ::=
<constraint type="world-state" relation="condition"
sender-id="">
    <parameters><list>
        <pattern-assignment>
            <pattern><list>
                <string> speed </string>
                <symbol> wind </symbol>
            </list></pattern>
            <value>
                <string> 35km/h <string>
            </value>
        </pattern-assignment>
    </list></parameters>
</constraint>
```

4.4. Annotations (A)

A is the set of annotations attached to the current plan. Amongst other things, annotations can be used to add human-centric information to the plan. This means, information that supports the human understand about the rationale of the plan. Annotations may be informal or they may adhere to some detailed syntax (which is not specified as part of <I-N-C-A>).

Annotations can be used to record arbitrary information about the plan (and the annotations form a part of this plan – hence the plan becomes, in some sense, self-descriptive). Specifically, we can see the annotation of plans with one particular type of rationale, namely the rationale information that can be recorded by the planner during the planning process. In this case, an annotation will be a syntactic expression of the form l_a:$r(l_p$:O, l_m:M, $O_1,\ldots,O_n)$, where:

- l_a: is a unique label for this annotation;
- r is a rationale predicate relating a plan-space object to other plan-space objects;
- l_p:O is a labelled plan-space object that is part of the current plan, i.e., it is an issue, an activity, a constraint or an annotation;
- l_m:M is an issue that was formerly in the plan and has since been resolved, i.e., it is a primitive meta-level activity that has been performed by the planner;
- $O_1,\ldots,O_n$ are plan-space objects that may or may not be labelled.

An annotation of this type represents the fact that the planspace object O was introduced into the plan as part of performing the plan modification activity M, and possibly involving other plan-space objects $O_1,\ldots,O_n$. The rationale predicate r denotes the relationship between these objects and describes the justification for including O. Thus, the interpretation of such an annotation depends on the rationale predicate r used. The different labels are necessary to specify the exact object that is being referred to. This is necessary as

there might be two activities in the plan which are identical except for the label. Several examples of annotations can be seen in[28].

5. A Unified Representation

This section discusses three different aspects related to planning process for coalition support systems. Each of these aspects has a set of requirements that must be considered, in a unified way, using <I-N-C-A> as basis for integration. The requirements were based on investigations of several coalition support systems and our experience during the development of some of them[19,27].

5.1. Planning agent aspects

5.1.1. Temporal model

The development of an appropriate temporal model, for planning in hierarchical coalitions, should be based on the following requirement to define the set *C1* (Figure 1) of constraints:

- Requirement 1: *the temporal planning model must be based on an explicit timeline approach, which* must *enable the representation of both quantitative and qualitative temporal references as well as relations between them.*

There are several and expressive ways that this requirement could be implemented[1,7]. We are using a set of timeline ideas to show how a temporal model could be specified via <I-N-C-A>. Considering the <I-N-C-A> representation, an explicit timeline approach indicates that each activity (node) has associated a constraint *I*, expressing its interval, with initial (*Ii*) and final (*If*) moments. Such a constraint could be defined as shown in Figure 4, where the relation attribute is set as *interval*.

For this type of constraint, we are composing the pattern, in the PATTERN-ASSIGNMENT element, by the node identifier; while the value is composed by the tuple <*Ii*,*If*> where *Ii* and *If* can be variables (identifiers starting with the ? *symbol*) if the moments are unknown. Based on this definition, instances of pattern-assignment for temporal constraints could be specified as:

```
CONSTRAINT ::=
<constraint type= "temporal" relation="interval"
sender-id= "ID">
    <parameters><list>
        <pattern-assignment>
             <pattern><symbol> node-x □/symbol□
</pattern>
             <value><list>
                  <integer> 10 </integer>
                  <integer> 20 </integer>
             </list> </value>
        </pattern-assignment>
        <pattern-assignment>
             <pattern><symbol>node-y </symbol></pattern>
             <value><interval>
                  <integer> 0 </integer>
                  <item-var> ?moment </item-var>
             </interval> </value>
        </pattern-assignment>
    </list></parameters>
</constraint>
```

Figure 5 illustrates a scenario where we can exemplify the use of this model to represent the temporal aspects of hierarchical coalition activities.

In this example, "East" is the region where the fires *F*1 and *F*2 are taking place. μ6 represents the command and control centre (strategic level); μ5 and μ4 represent the police office and the fire station respectively (operational level); μ3, μ2 and μ1 represent one police force and two fire brigades respectively (tactical level). Using the "relation-attribute(parameter)" notation, the following constraints can be specified for each of the activities in the strategic and operational levels:

- Plan of μ_6: overlaps(N_1,N_2);
  - N_1: interval(N_1,(0,?a));
  - N_2: interval(N_2,(?b,?c));
- Plan of μ_5: before($N_{1.1}$,$N_{1.2}$);
  - $N_{1.1}$: interval($N_{1.1}$,(0,?d));
  - $N_{1.2}$: interval($N_{1.2}$,(?e,?f));
- Plan of μ_4: finishes($N_{2.2}$,$N_{2.1}$);
  - $N_{2.1}$: interval($N_{2.1}$,(?g,?h));
  - $N_{2.2}$: interval($N_{2.2}$,(?i,?j));

Note that it is very difficult to determine durations for activities related to disaster relief operations. Thus their initial and final moments are likely to be variables.

This time representation allows the expression of values for both certain (with numeric values) and uncertain (with variables) times. In case of certain time, the duration of an activity can directly be defined as the difference between the final moment and initial moment. Considering now that we want to set temporal relations between two activities *a1* and *a2*, with respective intervals *I*(*a1*) and *I*(*a2*). The representation of temporal relations via <I-N-C-A> follows the structure shown in Figure 4, however with the relation

```
CONSTRAINT ::=
<constraint
type="resource"relation="RES-TYPE"sender-id="ID">
  <parameters><list>PATTERN-
ASSIGNMENT</list></parameters>
  <annotations><MAP> MAP-ENTRY </map></annotations>
</constraint>
```

Figure 4. Temporal constraint definition.

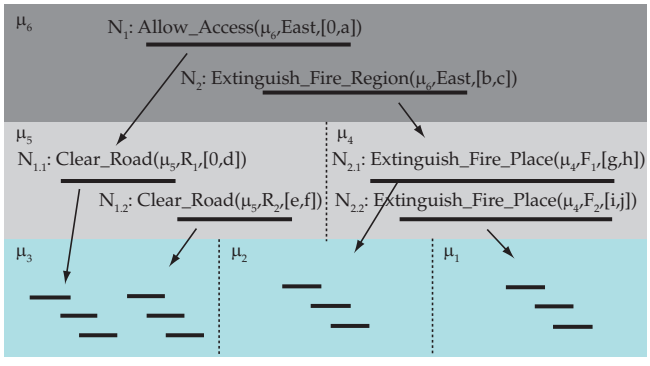

Figure 5. Example of activities and their intervals in a coalition.

attribute specifying a temporal relation (before, equals, meets, etc.) and a simple tuple (*a1*, *a2*) as parameter rather than a PATTERN-ASSIGNMENT element. The symbols *a1* and *a2* are the identifiers of the nodes (activities) that are being related. Then, to specify a temporal constraint whose semantics is *acitivity1* before *acitivity2*, we have:

```
CONSTRAINT ::=
<constraint type= "temporal" relation="before" sender-
id= "ID">
    <parameters><list>
         <symbol> activity1 </symbol>>
         <symbol> activity2 </symbol>>
    </list></parameters>
</constraint>
```

We can conclude that this simple representation for *C1* supports the Requirement 1. First it considers initial and final moments to activities so that they have explicit timelines. Second we can represent the notion of qualitative time, using temporal relations, and also quantitative values to express duration of activities. It is interesting to observe that temporal relations, such as "before", are abstractions on numeric relations between initial and final activities' moments. A more expressive representation could enable relations on any two activities' moments, as for example, to specify exact overlap periods between activities. However, this approach increases the manipulation complexity of such a representation so that its implementation could not be justified.

5.1.2. Resource model

The requirement used as a basis for the resource model specification via constraints (*C2* in Figure 1) is:

- Requirement 2*: the resource planning model must support the tasks of localising services/agents that provide specified capabilities, and also provide information that enables reasoning on such capabilities.*

The original approach of <I-N-C-A> uses a "pattern" in the activity element definition[26] to specify which capability an agent should have so that it could carry out a specific activity. For that end, the pattern is composed of an initial verb followed by any number of parameters, qualifiers or filler words. For example: (transport ?injured from ?x to ?hospital). Then, the system finds agents to perform this activity by matching the verb "transport" with the capabilities (list of verbs) of available agents. This simple mechanism supports the task of localising agents.

According to Requirement 2, the resource description should also provide information that enables reasoning on such capabilities. Imagine the scenario where a high building is on fire. To extinguish the fire in this building, the fire brigade should have a suitable ladder to reach the fire. However, using this simple capabilities description, fire brigades with and without ladders can be allocated to this activity because both are able to extinguish fires. Based on this fact, we can define a new type of constraint (Figure 6) to represent the features of a required resource as:

This constraint specification employs the same structure of the world-state or temporal constraints specification.

```
CONSTRAINT ::=
<constraint
type="resource"relation="RES-TYPE"sender-id="ID">
  <parameters><list>PATTERN-
ASSIGNMENT</list></parameters>
  <annotations><MAP> MAP-ENTRY </map></annotations>
</constraint>
```

Figure 6. Resource constraint definition.

Again we must detail the general form of the PATTERN-ASSIGNMENT element, which is defined as:
((resource object [resource-qualifier][resource-range]) value)

In this statement, "object" represents the type of the agent that accounts for performing the activity. In some cases, "object" can represent the identifier of an agent if we want to force that a specific agent has a specific status. For example, if the activity is "Extinguish ?fire", the object could be a "fire-brigade-x". The attribute "resource" represents some object's resource necessary for its operation (e.g. water-tank). Such resource can be qualified via the attribute "resource-qualifier" (e.g., quantity) and it can also have a range (e.g., from 0 to 10000 L), which is typically used when the resource is consumable.

Using this approach, the resource model can be seen in two levels. The activity pattern provides a simple high-level description of the capability required by the plan, while the constraints provide a more granular way to characterize or restrict the use of such capability. Considering a hierarchical coalition, for example, superior agents can keep only the high-level description of their subordinate agents, mainly because such descriptions are stable. However, if they need more information, a query can be performed so that subordinates return their current resource attributes and respective values. This interaction between agents evinces the influence that the resource specification of subordinates has on the planning process of their superior agent.

5.2. Teamwork aspects

5.2.1. Commitment and report function

The temporal and resource models, defined via<I-N-C-A>, provide the essential requirements for the development of multiagent planning processes, also providing the basis for the definition of more detailed models. As our approach is considering a multiagent organization based on a hierarchical structure, coordination is carried out by central agents, which account for developing incomplete plans and delegating tasks to their subordinates so that they can complete such plans. However, this notion of coordination is not enough to ensure that agents work together as a team.

Considering such facts, our next aim is to incorporate the notion of teamwork collaboration[4] into these planning processes. For that end, two initial requirements are considered during the specification of the commitment/report function (*F1*, Figure 1):

- Requirement 3: *the collaborative model must consider the establishment of commitments to joint activities, enabling consensus on plans or their constituent parts.*
- Requirement 4: *the collaborative model must provide ways to the dissemination of information associated with progress, completion and failure of activities.*

These requirements are considered in several works related to Teamwork such as *Joint Intentions*[22], *SharedPlan*[9], Joint *Responsibility*[12] and *Planned Teams*[13]. In our case, we are integrating planning and teamwork ideas via an algorithm (*F1*) that considers the plan creation as one of its steps. For that end, consider that such an algorithm is carried out by an agent μ, member of a coalition θ_x, that receives an activity p_i from its superior agent *sender*. This algorithm is codified via the "CollaborativePlanning" function (Figure 7).

This function entails some implications. First, the function tries to generate a *subplan* to perform p_i (step 02). If a *subplan* is possible (step 03) and it does not depend of anyone else (step 04) then the agent can commit to p_i (step 10). However, if *subplan* depends on the commitment of subordinates, then μ must delegate the necessary nodes to its subordinates and wait for their commitments (step 05). This means that commitments are done between a superior agent and their subordinates and, starting from the bottom, an "upper-commitment" can only be done if all the "down-commitments" are already stabilised.

Second, if some subordinate agent is not able to commit (step 06), μ returns (step 07) to generate other *subplan* rather than sending a failure report to its superior. Such a situation is similar to the cases where *subplan* is violated or μ receives a failure message of its subordinates (step 15). This approach implements the idea of enclosing problems inside the subteam where they were generated. Third, if μ is not able to generate a *subplan* for p_i, it reports a failure to its superior (step 21).

```
01. function CollaborativePlanning(sender, p_i)
02.    subplan ← PLAN(μ, p_i)
03.    if(∃subplan)
04.       if(hasNodesToBeDelegated(subplan)) then
05.          Delegate(subplan,subordinates)∧WaitCommits()
06.          if ∃s(s ∈ subordinates)∧(¬ commits(s)) then
07.             go to step 02
08.          end if
09.       end if
10.       Report(sender, p_i, commited)
11.       Broadcast(θ_x,subplan,conditions)
12.       while(¬ Complete(subplan))
13.          if(JustReady(subplan)∨Changed(subplan))then
14.             Report(sender, p_i,executing)
15.          else if(Violated(subplan)∨Receive(failure))
then
16.      go to step 02
17.          end if
18.       end while
19.       Report(sender, p_i, completion)
20.    else
21.       Report(sender, p_i, failure)
22.       ∀s (s ∈ subordinates)∧ HasCommitted(s,subplan_s)
23.          Report(s, subplan_s,failure)
24.    end if
25. end function
```

Figure 7. Collaborative planning algorithm.

In addition, it must also alert their subordinates that p_i has failed and consequently its subnodes can be abandoned (steps 22 and 23). Fourth, if conditions of p_i are changed, so that the task p_i becomes meaningless, the agent sender of p_i will generate failure reports and send such reports to all involved agents. The interpretation for this situation is the same when *subplan* fails. Remember that p_i in the level n+1 is part of *subplan* in an upper level n.

After reporting a commitment (step 10), μ must monitor and report execution status until the completion/failure of p_i. Progress reports are associated with changes in the plan, which are monitored and sent to superior as an ongoing execution report (step 13). Constraint violations and failure messages are also monitored (step 15) so that μ firstly tries to repair the problem by itself (step 16) before sending a failure report. Using this function, any activity *p* will have one of the following status: *not-ready*, *possible*, *impossible*, *complete* and *executing*. The <I-N-C-A> definition for activities contains a status attribute that can be filled with one of these options.

According to the Joint Intentions Theory[22], if μ finds out a problem in *subplan*, all the commitments previously associated with *subplan* should be cancelled. We are following this approach, so that we replan the full original task (step 2). Differently, the Joint Responsibilities Theory[12] states that if θ_x becomes uncommitted to subplan, there may still be useful processing to be carried out. In other words, the replanning should take into advancement of the conditions that resulted from the part of the plan that was performed before replanning. This latter approach seems to be more efficient and it is a likely research topic for our future extensions.

5.2.2. Mutual support

The next requirement, related to teamwork, leads the implementation of *F2* (Figure 1) and can be defined as:

- Requirement 5: *the collaborative model must underline the idea of mutual support, providing mechanisms for useful information sharing and allowing agents to create activities that support the task of other agents.*

The principal idea behind mutual support is to enable that one agent has knowledge about the needs of other agents. For example, μ knows that a specific road is clear so that it uses this constraint in its plan. However, as the world is dynamic, the road becomes blocked. If any other agent finds out that such road is no longer clear, it must inform this fact to μ. Thus, this informer agent is supporting the performance of μ.

An easy option to implement this model of knowledge sharing is to force that agents broadcast any new fact to all coalition. Consequently all agents will have their knowledge base updated and problems like that can be avoided. However, this is not a good approach in terms of communication and agents will also receive much useless information.

Consider now that the *subplan* of μ ($\mu \in \theta_x$) has a set of conditional constraints *C* (*C6* in Figure 1), which μ desires to hold so that its *subplan* is still valid. In this case, each $ci \in C$ is a constraint that μ believes to be true and hopes that it is still true. Then μ broadcasts *C* (step 11, Figure 6) for every agent μ_j

$\in \theta_x$ so that other agents of its subteam know what it needs. A function based on this idea (*F2* in Figure 1), and applied by agents that receive *C* from μ , is defined in (Figure 8).

According to the function, each agent μ_j must compare its beliefs BEL(μ_j) with *C* (step 03). If μ_j finds some *conflict*, it must try to create a new activity whose goal is to turn c_j true (step 04). If this is not possible, μ_j must inform μ that c_j is no longer holding and its new value is c_k (step 05). The idea implemented by this function is simple, however there are two more complex points: the "Conflict" and "Valid" functions.

The Conflict function (step 03) is an extension of the Violated function (step 15, CollaborativePlanning function – Figure 7). A violation is a type of conflict between two constraints. It says that two constraints, which are supposed to match, are not matching. However we are also considering as conflict the situation where two constraints have the potential to be identical. For example, ((colour Car),?x) and ((colour Car),blue). In this case, the two constraints are in conflict because they have the potential to be identical if the variable ?*x* assumes the value "blue".

This type of conflict is very useful in the following class of situations. Suppose that one of the activities of μ is to rescue injured civilians. For that end, μ firstly needs to find such civilians so that it has the following conditional constraint:

```
CONSTRAINT ::=
<constraint type="world-state" relation="condition"
sender-id="">
    <parameters><list>
        <pattern-assignment>
            <pattern><list>
                <string> position </string>
                <item-var> ?a </item-var>
            </list></pattern>
            <value>
                <item-var> ?b </item-var>
            </value>
        </pattern-assignment>
        <pattern-assignment>
            <pattern><list>
                <string> role </string>
                <item-var> ?a </item-var>
            </list></pattern>
            <value>
                <string> civilian </string>
            </value>
       </pattern-assignment>
       <pattern-assignment>
            <pattern><list>
                <string> status </string>
                <item-var> ?a </item-var>
            </list></pattern>
            <value>
                <string> injured </string>
            </value>
        </pattern-assignment>
    </list></parameters>
</constraint>
```

The list of parameters of this constraint can be summarized as: ((position ?*a*),?*b*), ((role ?a),civilian) and (status ?*a*),injured). Such list implies that the variable ?*b* is the location of an injured civilian ?*a*. Let consider that another coalition agent has or discovers the following propositions: ((position James),(45.6,78.9)), ((role James),civilian) and (status James),injured). Then, this agent has the knowledge to match the variables ?a and ?b of μ. Consequently, the agent must inform μ about this new knowledge (note that in this case it does not make sense to create a new activity).

```
01. function MutualSupport(μ, C)
02.    while (∃c_j  c_j ∈ C)
03.       if(∃c_j c_k c_j ∈ C ∧ c_k ∈ BEL(μj)∧Conflict(c_j,c_k))then
04.             newactivity ← CreateActivity(Goal(c_j))
05.             if(¬∃ newactivity)then Inform(μ,c_k) end if
06.             Retire(c_j, C)
07.          end if
08.          if (∃c_j  c_j ∈ C ∧ ¬Valid(c_j)) then
09.             Retire(c_j, C)
10.          end if
11.       end while
12. end function
```

Figure 8. Mutual support algorithm, based on Requirement 5.

The Valid function (step 08) accounts for eliminating the constraints that no longer represent conditions to μ. This is important to avoid that μ still receives useless information and also to decrease the number of messages in the coalition. A practical way to do that is to consider that all $ci \in C$ has a timestamp that indicates the interval where such constraint is valid.

Using the timestamp (t_i ,t_f) and considering that t_i and t_f are ground values, the Valid function only needs to compare if the condition ($t_f <$ current-time) is true to eliminate the respective constraint. However these timestamps are not useful if agents do not know when their activities finish because such a temporal value will be a variable. Note that the principal advantage that we are looking for in using timestamps is to avoid that agents (C's senders) need to broadcast the information that they no longer need that a group of constraints holds. Rather, timestamps enables agents (C's receivers) to reason by themselves on the elimination of such constraints. An alternative for timestamps, which we intend to investigate, is to link constraints to the partial ordering of the plan elements rather than real world time.

One of the principal advantages of the *MutualSupport* function is that it improves the information sharing in θ_x because the sending of information is guided by the constraint-based knowledge that each agent has about the activities of its partners. In addition, it can also be used as a method to avoid conflict between activities because agents know which external constraints must be respected.

5.3. Human interaction aspects

5.3.1. Agents' autonomy

The use of teamwork ideas supports the performance of collaborative planning activities by agents of a coalition. However such ideas do not consider situations where agents interact with human users. A first problem that could be raised in such situation is that the agent inaction while waiting for a human response can lead to potential misco-

ordination with other coalition members[3]. This problem, in particular, enforces the requirement in follow:

- Requirement 6: *the human-agent model must enable the definition of adjustable methods that complement the decision making process of human users.*

The transfer of control between agents and humans, in our architecture, follows a *generate-evaluate-choose* sequence. First an agent generates possible plan options to deal with the current set of activities, presenting such options to its users. Then users evaluate these options and, if they want to make changes, the process returns to the first step where its agent generates a new set of options. Otherwise, users choose one of the options to be performed.

In this scenario, an interesting research issue is related to the support that an agent provides to its human user, so that an option can be chosen. We have seen, during the discussion of <I-N-C-A>, that this ontology has a component called "Annotation". Annotations can save the rationale of the plan, indicating, for example, plan metrics such as required time, and reasons for decisions. A step toward this approach is discussed latter on (*Explanation Function*).

Independently of the support via annotations, the *generate-evaluate-choose* approach is very useful to combine the abilities of humans and agents so that they synergistically work together. We can say that while users have the ability to take decisions based on their past-experiences (case-base reasoning), agents are able to generate and compare a significant number of options, showing both positive and negative points of such options.

Unfortunately, this process also can generate delays in the planning process, due to the human reaction/response time. An option to decrease the impact of human delays in this process is to define contexts where agents choose by themselves the option to be performed. Such an approach was used in past projects[15] and it is very suitable because contexts can be redefined by users so that they are still in control of the situation.

In our approach, we are defining contexts associated with activities so that if such a context is true, a degree of autonomy is applied avoiding or allowing agents to take autonomous decisions. Constraints (C_8 in Figure 1) that implement this idea have their type attribute instantiated with the keyword *autonomy*, and the relation attribute with a *degree* representing the agents' level of autonomy (Figure 9).

```
CONSTRAINT ::=
<constraint type="autonomy" relation="DEGREE"
sender-id="ID">
  <parameters><list>PATTERN-
ASSIGNMENT</list></parameters>
  <annotations><MAP> MAP-ENTRY </map></annotations>
</constraint>
```

Figure 9. Autonomy constraint definition.

Based on this explanation, consider the following example of constraint:

```
<constraint type = "autonomy" relation="permission">
     <parameters> <list>
          <pattern-assignment>
               <pattern><list>
                    <string> speed </string>
                    <symbol> Wind </symbol>
               </list></pattern>
               <value> <list>
                    <symbol> greater-than </symbol>
    <string> 20mph </string>
               </list></value>
          </pattern-assignment>
    <pattern-assignment>
               <pattern><list>
                    <string> water-tank </string>
                    <symbol> FireBrigadeOne </symbol>
               </list></pattern>
               <value> <list>
                    <symbol> less-than </symbol>
    <string> 6000000mm </string>
               </list></value>
          </pattern-assignment>
     </list></parameters>
</constraint>
```

This constraint means that if the wind speed is greater than 20 mph and the amount of water in the FireBrigade1 tank is less than 6000000 mm, then the activity associated with this constraint must ask permission to be performed by the agent. Note that such a constraint tries to configure an inappropriate scenario (low amount of water and violent wind) for fire brigade performance. Then the agent must pass the final decision to its human user.

Currently we are considering two degree values for these constraints: *permission* and *in-control*. The first is the default value so that if no autonomy constraint is specified, agents consider that they have to ask permission to insert (and perform if the agent is also an executor) the associated activity. The second means that agents can choose and perform the activity by themselves. The PATTERN-ASSIGNMENT element is defined as ((*attribute object*), *value*) so that a list of pattern assignments creates a context for possible values where the degree must be activated.

A possible expansion of this approach is to implement the *consult* (degree attribute) autonomy constraint. The idea is to enable that users specify context where agents must consult humans about the instantiation of specific variables. In terms of representation, extensions for this approach should consider the *consult* keyword as an option for the DEGREE parameter and find ways to indicate which planning variables are involved in this process.

Note that using *permission* constraints, agents only ask humans to confirm the choice of one activity. Differently, using *consult* constraints, the process becomes more interactive and granular because humans interfere during the configuration of activities. Later on we will see that the *consult* degree is not actually needed because the model already considers user preferences on instantiations of variables. However we have a slight difference: while the use of *consult* constraints becomes the process dynamic (users must decide for a value at runtime), the use of preferences is static because all the preferences are pre-defined by users before the own execution.

5.3.2. Users restrictions

The second requirement, associated with the involvement of human users during the collaborative planning process, tries to avoid problematic local decisions. Note that local decisions taken by a coalition member can seem appropriate for her/him, but may be unacceptable to the team. Based on this fact, we should have the following requirement:

- Requirement 7: *the human-agent model must provide ways to restrict user options in accordance with the global coalition decisions.*

The idea here is to avoid that users take decisions that are prejudicial to the coalition as a whole. For example, an user could make the decision of using a road as save point for injured civilians. However this road is going to be used as access route of a fire brigade. Thus, the former decision must be avoided.

Agents can do that by restricting the options of users in creating their plans, or performing their activities. The natural way to implement such restrictions, considering our constraint-based framework, is to define a set of constraints (C_5 in Figure 1) that cannot be changed by users or their planning agents. During the planning process, users are able to manipulate constraints as a way to customise the outcome solutions, as discussed in the next section. However, it is clear that there is a group of constraints that users cannot change. This group is represented by constraints whose source is an external component such as the superior agent, or other team members. Based on this idea, the whole set of constraints can be divided into two classes:

- *Internal constraints*, which users can manipulate (read/write) and that are generally created by themselves during the process of customisation of solutions;
- *External constraints*, which users cannot change (read-only) because they are assigned by other components.

<I-N-C-A> enables a very direct support for such classification. As we can see in the constraint definition (Figure 3), constraints have an optional attribute (*sender-id*) that specifies the constraint source, or the component that accounts for their creation. Thus the semantics associated with such attribute is very simple. If the *sender-id* attribute is assigned with the own agent/user identifier, the constraint is internal. Otherwise the constraint is external.

Despite the simplicity of such approach, it also presents a serious limitation. For example, suppose that a superior agent sends an activity to one of its subordinates. Associated with this activity, the superior agent also sends a group of constraints with some preferences on how to perform such activity. Note that these constraints are only preferences so that the subordinate agent is not obliged to follow the options expressed by such constraint group. It must do so if it is able to. Otherwise it can try other options. However this is not possible using our initial approach. As the constraints are external, they cannot be changed and planning must respect them.

This feature can be relaxed by implementing the idea of weak constraints. Weak constraints are, at first, used as normal constraints. However, if agents are not able to find any solution, they can eliminate weak-constraints, relaxing the restrictions on the plan options. Following this approach, world-state, resource and temporal constraints should also have their weak versions. Thus, the ontology needs to have ways to discriminate between normal and weak constraints. The next section shows how this discrimination is specified in our model.

5.3.3. Users control

The next requirement to be considered is:

- Requirement 8*: the human-agent model must support the definition of mechanisms that intensify the human user control and enable the customization of solutions.*

This requirement is traditionally investigated in *Mixed-Initiative Planning* projects, such as in TRANS[6] and O-Plan[25]. Note that it seems to be antagonistic to Requirement 7. In fact we are looking for a mutual process of restriction so that while agents have constraints (C_5 in Figure 1) to restrict the options of human users, users can also set constraints (C_4 in Figure 1) to restrict the behaviour/reasoning of agents.

The approach defined above provides the basis for user control. According to that approach, users can set normal constraints to force some result, or weak constraints to try such preferential solution if they are possible. To implement this approach (weak constraints) we have defined a new class of constraints (C_4) whose type attribute is specified as *preference* (Figure 10) and the relation attribute receives the type of a pre-defined constraint such as temporal or resource type constraints.

A more practical way to deal with preferences is to associate them with the process of variable instantiation. Consider that such instantiation can be guided by a specific kind of constraint, whose function is to restrict values of constraint variables. This new type of preference constraint has its relation called *instantiation* and its PATTERN-ASSIGNMENT element defined as: ((*set-definer-attribute variable*), *value*). Possible examples are:

- ((> ?x), 10) – indicates preference for values greater than 10;
- ((set ?r), R_5) – this is the most basic example for discrete values, where ?r should be instantiated with the value R_5;
- ((in ?r), [R_4, R_3, R_5]) – in this case ?r should be instantiated with any value that belongs to the set [R_4, R_3, R_5];

```
CONSTRAINT ::=
<constraint type="preference" relation="PREF-TYPE"
sender-id="ID">
 <parameters><list>PATTERN-
ASSIGNMENT</list></parameters>
  <annotations><MAP> MAP-ENTRY </map></annotations>
</constraint>
```

Figure 10. Preference constraint definition.

- ((in-ordered ?r), [R_4, R_3, R_5]) – in this case ?r should be instantiated with any value that belongs to the set [R_4, R_3, R_5], but considering that the values are listed in order of preference.

The last example is particularly important because it demonstrates the expressiveness of the syntax. In this case the user is not only setting a specific preference, but a set of discrete and disjunctive values listed by order of preference. Then if R_4 is not possible, the agent must firstly try the next option that is R_3 and so on. Based on these examples, the idea of set-definer-attribute can be expanded to consider diverse kinds of delimiters.

Note that autonomy and instantiation/preference constraints have different semantics for their parameters, if we compare them with the other constraint types. While the autonomy constraint parameter characterises a context in which a specific degree of autonomy is valid, the instantiation/preference constraint parameter indicates preferential values to be assigned to variables.

5.3.4. Explanation function

The last requirement that must be considered is:

- Requirement 9*: the human-agent model must support the generation of explanations about autonomous decisions, clarifying the reasons why they were taken.*

This requirement is important because human users have a strong need of understanding what and why something is happening or will be carried out by the agent, mainly when critical decisions must be taken. In this way, the idea here is to define a function (F_3 in Figure 1) that produces explanations based on specific events triggered during the constraint processing. For example, if an activity becomes impossible due to a constraint, this event generates an explanation to the user saying which constraint(s) is blocking the plan. In this case the user can, for example, relax or delete such constraint (only if it is an internal constraint). In this case the set of constraints C_7 associated with F_3 represents all constraints that are being manipulated by the planning process. Considering this idea, we must answer three questions[20]: which are the agents' decisions that humans would like to have explanations for, which are the events that can trigger such explanations, and how can such explanations be produced.

To exemplify the explanation function design, consider one of the most common events that happens during the planning and execution process: the invalidation of an activity in a plan. Users are generally interested to know the reasons for such invalidation. In other words, explanations for this case must be able to answer questions like: "why is the plan/activity *x* impossible?" or "why are we not able to perform *x*?".

The second step is to identify the constraint processing events that can be used to generate explanations for such questions. We know that the invalidation of plan options is caused by conflict between constraints. Considering that the plan is in a stable state, some event must happen to cause a conflict. Such an event could be: activity addition, state changes and user decisions.

The approach designed to produce explanations it to use templates with variables, which are instantiated by such events. The idea is to translate the meaning of each event via a specific template. A template for the case of activity invalidation is: *activity* is not valid because $c_{activity}$ is in conflict with $c_{trigger}$ set by *source*.

For this template, F_3 must instantiate four variables: *activity* representing the activity that was invalidated, $c_{activity}$ representing the activity constraint in conflict, $c_{trigger}$ representing the constraint that has triggered the conflict and *source* representing the source of $c_{trigger}$. Generalising the idea used in this example, we can define F_3 as follows (Figure 11).

The idea of this function is to monitor the activity/plan *p* while it is valid. Then if some event associated with the constraint processing of *p* happens, the function captures such event (step 03) and, if it is an explanation trigger, the function gets a predefined template for it (step 05). Then the variables of this template are instantiated in accordance with several features (type, sender-id, constraint associated, etc.), creating the explanation for this event. Finally the function sets the explanation to *p* (step 07).

The explanation function defined here is able to generate high-level explanations for simple decisions taken by agents. However, the idea provides the basis to support the elaboration of more detailed explanations for questions like: "Why has the plan/activity *x* been chosen?", "Why is this solution better than that?" or "Which are the possible values for a specific domain variable?". Note that while an explanation function must capture the notion of comparative parameters in the first two questions, the last question requires a better understanding of the planning domain.

6. The Implementation Approach

The main role of assistant planning agents is to provide actions to decompose nodes until there are only executable nodes. For that end, agents consider planning as a two-cycle process, which aims to build a plan as a set of nodes (activities) according to the <I-N-C-A> approach. The first cycle tries to create candidate nodes to be included into the agent's plan, respecting the current constraints of the activities, which are already in the plan. If the agent is able to create one of more candidate nodes, one of them can be chosen and

```
01. function MutualSupport(p)
02     while (Valid(p))
03         event ← CaptureEvent(p)
04          if (∃ event) then
05              template ← GetTemplate(event)
06              explanation ← InstantiateTemplate(template,
                                  GetFeature(event))
07               SetTemplate(p,explanation)
08           end if
09      end while
10 end function
```

Figure 11. Preference constraint definition.

its associated constraints are propagated, restricting the addition of future new nodes.

A simple way to understand this process is to follow the example below (Figure 12). The current agent's plan contains the set of activities that it intends to perform. If the agent receives a new activity, it must generate planning *actions* that include this new node in its plan. Each action is a different way to perform this inclusion so that different actions generate different nodes' configurations.

We call this process of activity-oriented planning because agents provide context sensitive actions to perform activities in specific. Common types of actions implemented under this perspective are:

- Delegation: action that simply sends a node to other agent that has the capability to handle it. Thus the unique work of the sender agent is to wait for the result;
- *Standard Operating Procedures* (SOPs): SOPs are sequences of pre-defined activities based on experiences, lessons learnt or carefully pre-designed. Depending on the context, agents turn available one or more SOPs as actions that decompose nodes;
- Dynamic Plan Generation: this action creates a dynamic plan, providing more assistance with a "How do I do this?" action associated with each node;
- Specific solver: actions can invoke specific solvers, such as a pathfinder, which are available as plug-ins.

The role of agents is to provide actions to decompose nodes until there are only executable nodes. The important point in this discussion is to know that each action is implemented by an activity handler, which propagates the components through constraint managers to validate their constraints. For example, the action of applying a SOP is a handler that decomposes an activity according to the SOP specification. For that end, the handler also causes constraint managers to check the conditions in which the SOP can be applied, indicating any conflict.

All agents have a set of activity handlers that they use to refine or perform their activities. In a general way, the process follows the steps in follow:

(i) When an activity a is received, the agent's controller component selects a set H of activity handlers, which matches the description of a;

(ii) Each handler $h \in H$ uses one or more constraint managers to return its status (possible, impossible or not ready);

(iii) Users choose one of the proposed handlers, committing to the performance of a;

(iv) During the execution, constraint managers are still monitoring the constraints of a, warning in case of problems.

The role of constraint managers in this process is to maintain information about a plan while it is being generated and executed. The information can then be used to prune search where plans are found to be invalid as a result of propagating the constraints managed by these managers. The principal advantage of using constraint managers is their modularity. We can design managers to deal with specific types of constraints, such as the types discussed here (e.g., temporal, resource, commitment, etc.).

Together, the constraint managers compose the model manager of the agent. Each constraint manager considers a set of specific constraints in a well-defined syntax, based on the <I-N-C-A> formalism. In brief, constraint managers provide support to a higher level process of the planner where decisions are taken. However, they do not take any decision themselves. Rather, they are intended to maintain all the information about the constraints they are managing and to respond to questions being asked of them by the decision making level[24].

Further details about this project such as published papers, example of application and related links can be found in the project resource page: "http://www.aiai.ed.ac.uk/project/ix/project/siebra/". From this page we can also download and use the I-Kobe prototype, an I-X appli-

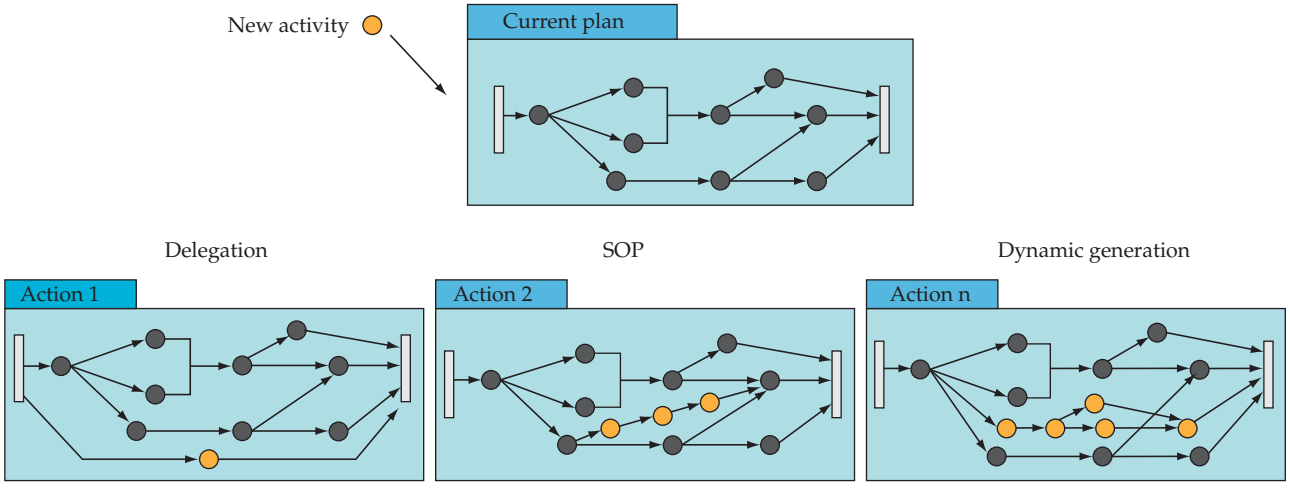

Figure 12. The activity-oriented planning approach.

cation to support disaster relief operations in the RoboCup Rescue Kobe scenario. The development of I-Kobe was based on the ideas presented in this paper so that we could demonstrate such ideas in a practical way. The I-Kobe manual gives details about the features and configuration of this application, while The QuickStart is a practical way of setting and running the application.

7. Related Works

Several systems have been developed with the aim of supporting the planning and execution performance of joint groups. This section summarises the main ideas of some of these systems, comparing such ideas with our approach.

The *CoAX Project*[2] demonstrates the use of the agent-based paradigm as a way to deal with the technical issue of integrating different technologies in a coalition organisation. The principal proposal of this project is to use agents to wrap different systems, enabling their integration via a common infrastructure. Differently, we have used the concept of ontology to integrate components. According to this approach, every external component that needs to be integrated to the system must respect ontological commitments when receiving and sending information to the coalition.

CplanT[16], a multiagent system that belongs to the area of war avoidance operations, developed a formal knowledge based approach to the coalition formation problem. The principal issue in this approach is that agents may agree to collaborate, but they are often reluctant to share their knowledge and resources. Thus, negotiation mechanisms are necessary to support the various levels of collaboration. In our approach we do not discuss any kind of negotiation process associated with information sharing. In fact, at the current stage, agents must share any kind of information that should be important to the performance of other coalition members. Negotiation is very suitable at pre-operation moments, when a coalition is involved in discussions related to which role will be played by sub-coalitions or members. However we are aware that in more complex kinds of coalitions, such as multinational coalitions, the idea of classifying and restricting the information access is appropriated and must be considered.

CODA[14] is a system that proposes to improve the coordination process using targeted information dissemination among distributed human planners. According to the CODA approach, each planner declares interest in different kinds of plan changes that could impact his/her local plan development. Thus, CODA is based on plan authoring tools, which are able to monitor the activities of users so that changes that match awareness are forwarded automatically to the person who declared interest in them. Our approach implements a similar idea, however without the need of users to declare which information they want. This information is directly extracted from the conditional constraints of each activity and analysed by the mutual support function. Similarly as happens in CODA, such a function provides an adequate way to share information, supporting conflict detection and resolution.

DSIPE[5] is a distributed planning system that provides decision support to human planners in a joint planning environment. DSIPE uses the same hierarchical structure for agents that we are using, however with a different approach to plan decomposition. In DSIPE each planning agent has a complete representation of its own subplan as well as a partial representation of the subplans being developed by other planning agents, with explicit dependencies and relationships with its local subplan. Thus the project implements a specific algorithm for information sharing where each agent knows the kind of information that could be important to other agents so that they can update their partial representations. The DSIPE approach certainly increases the system complexity because the filter algorithm needs to have specific information about other agents and their activities. However such algorithm could be interesting for us. Note that our proposal for mutual support is based on a broadcast of conditional constraints to a sub-coalition. Probably, some agents from this sub-coalition will not use these constraints in their process. Thus we could use a filter algorithm to change the broadcast for a peer-to-peer process.

APSS[11] is a proposal of decision support system that seeks to merge planning and execution, and replaces reaction to events with anticipation of events. For that, rather than choosing a single course of action (COA) and following it to conclusion, the system maintains many possible COAs so that the plan is considered to be a tree. Nodes of this tree represent states and decision points in the plan. The branches represent the transition to a new state based on a particular action. As new branches are developed, the system will continue planning along those branches. Thus, anticipatory planning for a branch can be done well in advance, rather than reactive planning once the branch occurs. The point stressed by the APSS project is the importance of the plan information collection to quickly confirm or deny the viability of branches. In our approach we do not try to create and maintain a tree with several branches, however we also consider fundamental the use of information to anticipate possible failures. Such an idea is mainly implemented via reports on execution progress, which try to capture the information that could support the reasoning of superior agents in detecting and resolving possible failures in a sub-coalition plan.

Note that such main projects in multiagent planning for coalition deal with specific problems, such as information sharing or plan decomposition. Differently our work proposes a framework that could integrate solutions for several coalition support requirements, rather than dealing with a specific issue. In some cases, the framework also presents alternative approaches for such specific issues, as discussed along this paper.

8. Conclusion and Research Directions

This work suggests a common way to consider several requirements related to the definition of a planning frame-

work for coalition support systems. In fact, several of these requirements are already investigated by other researches, however in an isolated way. Thus, their integration is not an easy task. The advantage of our approach is that the implementation of all requirements can be understood from a unique perspective and implemented on the same basis (using constraints representation and manipulation).

The main problem of our approach is derived from processes associated with the sharing and maintenance of distributed knowledge (e.g., persistence of no longer valid knowledge inside the coalition), as better discussed in[18]. In a more specific way, these problems are associated with our approach for mutual support. Thus, future investigations and extensions of this work include:

- Development of experiments that measure the usefulness and usability of conditional constraints, considering the process of mutual support. The idea is to investigate, from the set of all constraints received by an agent, which of such constraints are useful for the different processes provided by the mutual support approach (conflict resolution, information sharing and activity generation);
- Study and implementation of mechanisms that enable the elimination of knowledge which is no longer valid from the coalition. Rather than agents exchanging messages saying which information must be eliminated, agents should be able to reason about such elimination by themselves. An interesting metaphor is to think about this process as a garbage collection used for some object-oriented languages. In Java, for example, each virtual machine uses a specific rule (there are no longer any references to an object) to eliminate unnecessary objects. In the same way, we could implement some rule in each agent so that they eliminate unnecessary knowledge;

Another possible direction of this work could consider its extension to applications whose features are incompatible with our current approach. For example, our approach is not compatible with domains that require dynamic and autonomous change of roles, and domains that require negotiation during the process of activity delegation. Moreover, our proposal does not support a direct implementation of reactive behaviour. The closer mechanism to reactive behaviour that we have provided is the definition of SOPs (*Standard Operating Procedures*). Probably such pre-planned sequences of activities are not enough to cope with the dynamic of coalition domains.

Acknowledgments

Clauirton Siebra' and Natasha Queiroz' scholarships were sponsored by CAPES Foundation under process numbers BEX2092/00-0 and BEX1944/00-2. This material is based on research within the I-X project, led by Professor Austin Tate and sponsored by the Defense Advanced Research Projects Agency (DARPA) and US Air Force Research Laboratory under agreement number F30602-03-2-0014 and other sources.

The University of Edinburgh, UFPB and research sponsors are authorised to reproduce and distribute reprints and on-line copies for their purposes not withstanding any copyright annotation here on. The views and conclusions contained here in are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of other parties.

References

1. Allen J. Planning as Temporal Reasoning. In: *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*; 1991; Massachusetts, USA. p. 3-14.
2. Allsopp D, Beautement P, Bradshaw J, Durfee E, Kirton M, Knoblock C, Suri N, Tate A and Thompson C. Coalition Agents Experiment: Multi-Agent Co-operation in an International Coalition Setting. *IEEE Intelligent Systems* 2002; 17(3):26-35.
3. Bradshaw J, Boy G, Durfee E, Gruninger M, Hexmoor H, Suri N, Tambe M, Uschold M and Vitek J. Software Agents for the War Fighter. In: *ITAC Consortium Report*. Cambridge, MA: AAAI Press/The MIT Press, 2002.
4. Coheh P and Levesque H. Teamwork. *Nous, Special Issue on Cognitive Science and Artificial Intelligence* 1991; 5(4):487-512.
5. DesJardins M and Wolverton M. Coordinating a Distributed Planning System. *AI Magazine* 1999; 20(4):45-53.
6. Ferguson G, Allen J and Miller B. TRAINS-95: Towards a Mixed-Initiative Planning Assistant. *In Proceedings of the Third Conference in AI Planning Systems;* 1996. Meno Park, California: AAAI Press; 1996. p. 70-77.
7. Freksa C. Temporal Reasoning Based on Semi-intervals. *Artificial Intelligence* 1992; 54(1-2):199-228.
8. Grosz B. Collaborative Systems. *AI Magazine* 1996; 17(2):67-85.
9. Grosz B, Hunsberger L and Kraus S. Planning and Acting Together. *AI Magazine* 1999; 5(2):23-34.
10. Gruber T. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal Human-Computer Studies* 1995; 43(5-6):907-928.
11. Hill J, Surdu J, Ragsdale D and Schafer J. Anticipatory Planning in Information Operations. In: *IEEE International Conference on Systems, Man and Cybernetics;* 2000; Nashville, Tennessee, USA. p. 2350-2355.
12. Jennings N. Towards a Cooperation Knowledge Level for Collaborative Problem Solving. *In Proceedings of the Tenth European Conference on Artificial Intelligence;* 1992; Vienna, Austria. p. 224-228.
13. Kinny D, Ljungberg M, Rao M, Tidhar G and Werner E. Planned Team Activity. *In Proceedings of the Fourth European Workshop on Modelling Autonomous Agents in a Multiagent World;* 1992; Rome, Italy. p. 227-256.
14. Myers K, Jarvis P and Lee T. CODA: Coordinating Human Planners. In: *Proceedings of the Sixth European Conference on Planning;* 2001; Toledo, Spain.

15. Myers K and Morley D. Policy-based agent directability. In: Hexmoor H, Falcone R and Castelfranchi C (Eds.). *Agent Autonomy.* Kluwer Academic Publishers; 2003. p. 187-210.

16. Pechoucek M, Marik V and Barta J. A Knowledge-Based Approach to Coalition Formation. *IEEE Intelligent Systems* 2002; 17(3):17-25.

17. Pegram D, Amant R and Riedl R. An approach to visual interaction in mixed-initiative planning. In: *Proceedings of the AAAI Workshop on Mixed-Initiative Intelligence*; 1999; Orlando, Florida, USA. p. 15-23.

18. Siebra C. *A Unified Approach to Planning Support in Hierarchical Coalitions.* [PhD Thesis]. Scotland, UK: The University of Edinburgh; 2006.

19. Siebra C, Tate A and Lino N. Planning and Representation of Joint Human-Agent Space Missions via Constraint-Based Models. In: *Proceedings of the Fourth International Workshop on Planning and Scheduling for Space; 2004;* Darmstadt, Germany.

20. Sqalli M and Freuder E. Inference-Based Constraint Satisfaction Supports Explanation. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*; 1996; Portland, Oregon, USA. p. 318-325.

21. Sycara K. Multiagent Systems. *AI Magazine* 1998; 19(2):79-92.

22. Tambe M. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research* 1997; 7:83-124.

23. Tambe M. Multiagent and Agent-Human Teamwork: Theory and Practice. In: *IJCAI-03 Tutorial*. Acapulco, Mexico; 2003.

24. Tate A. Integrating Constraint Management into an AI Planner. *Journal of Artificial Intelligence in Engineering* 1995; 9(3):221-228.

25. Tate A. Mixed Initiative Interaction in O-Plan. In: *Proceedings of the AAAI Spring Symposium on Computational Models for Mixed Initiative Interaction*; 1997; Stanford, California, USA.

26. Tate A. <I-N-C-A>: an Ontology for Mixed-Initiative Synthesis Tasks. In: *Proceedings of the IJCAI Workshop on Mixed-Initiative Intelligent Systems*; 2003; Acapulco, Mexico.

27. Tate A, Dalton J, Siebra C, Aitken S, Bradshaw J and Uszok A. Intelligent Agents for Coalition Search and Rescue Task Support. AAAI-2004 Intelligent Systems Demonstrator. In: *Proceedings of the Nineteenth National Conference of the American Association of Artificial Intelligence*; 2004; San Jose, California, USA.

28. Wickler G, Potter S and Tate A. Recording Rationale in <I-N-C-A> for Plan Analysis. In: *Proceedings of the ICAPS Workshop on Plan Analysis and Management*; 2006; Lake District, England.

29. Wilkins D and Myers K. A Multiagent Planning Architecture. In: *Proceedings of the Forth International Conference on AI Planning Systems*; 1998; Pittsburgh, USA. p.154-162.

30. Wolverton M and Desjardins M. Controlling communication in distributed planning using irrelevance reasoning. *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*; 1998. p. 868-874.