Using the KGP Model of Agency to Design Applications

Fariba Sadri

Department of Computing,
Imperial College London,
180 Queen's gate, London SW7 2BZ, UK
fs@doc.ic.ac.uk

Abstract. This paper is a tutorial describing the main features of the KGP (Knowledge-Goals-Plan) model of agency and giving user guidance on how the model can be used to develop applications. The KGP model is based on computational logic. It consists of an abstract component, a computational component and an implementation. This paper concentrates on the abstract component, which consists of formal specifications of a number of different modules, including the knowledge bases, capabilities, transitions and control. For each of these we summarise what is provided by the model, and through the platform implementing the model, and what is left to the users to specify according to the requirements of the applications for which they wish to use the KGP model to design agents.

1 Introduction

1.1 The Model

The KGP (Knowledge-Goals-Plan) model of agency has been developed within the EU SOCS (Societies of Computational Entities) project in a collaborative effort involving Imperial College, City University, and the universities of Cyprus, Pisa, Bologna, and Ferrara. Information about the project can be found at http://lia.deis.unibo.it/research/socs/.

The model is general purpose and highly modular. All of its components, including its control component, are based on computational logic, and more concretely on abductive logic programming [8] and logic programming with priorities [3], both with extensions that deal with temporal constraints.

The model includes:

- an abstract model (declarative semantics): providing formal specifications, in computational logic, for all the components,
- a computational model (operational semantics): providing a computational counterpart for all the formal components of the abstract model, and
- a prototype implementation (PROSOCS) in Prolog, Java and JXTA [19].

The computational model exploits the modularity of the abstract model and has been proved correct with respect to it. It consists of:

F. Toni and P. Torroni (Eds.): CLIMA VI, LNAI 3900, pp. 165 – 185, 2006.

- a proof procedure, CIFF [4,5,6], that combines abduction and constraint logic programming, for the components of the model that are based on abductive logic programming, and
- a proof procedure, Gorgias [7], that combines argumentation and constraints, for the components of the model that are based on logic programming with priorities.

In this paper we concentrate on the abstract model.

The KGP model has been designed to cater for the needs of a global computing setting. To this end it provides heterogeneity, allowing agents to be designed such that they differ from each other in their knowledge and behaviour. It also incorporates features that allow agents to function in dynamic open environments, adapt to changes in the environment and interact with other agents.

The model integrates various aspects of agency, including:

- Reasoning: for example for planning and proactivity
- Reactivity: for example allowing agents to react to changes they perceive in their environment by performing actions, including sending communications to other agents
- Goal introduction: allowing agents to alter their goals according to their circumnstances
- Declarative control: providing dynamic control of the operations of the agent
- (some) Belief revision: for example allowing agents to modify their beliefs in the light of their observations
- Interaction: for example allowing agents to negotiate with one another for resources.

The model and its prototype implementation have been used in applications in combinatorial auctions and negotiation for resources. The formal basis of the model facilitates formal specification and verification of properties. Such properties have been studied and are reported in [1].

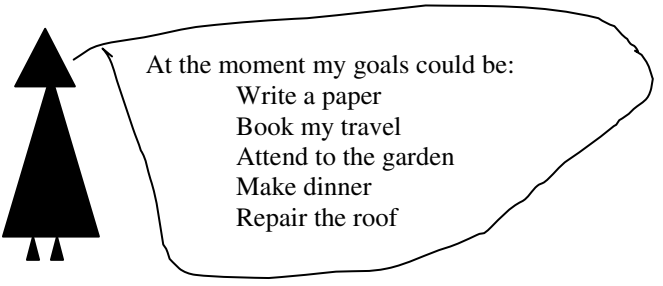
A detailed description of the KGP model and its comparison with other models can be found in [9,10], a summary in [2, 11], and details of its implementation in [18]. Details of some components of the model can be found in [14] for the planning component, in [12] for the control component, and in [4, 5, 6] and [7] for the proof procedures. Extensions of the model than incorporate normative concepts can be found in [16, 17].

1.2 Examples

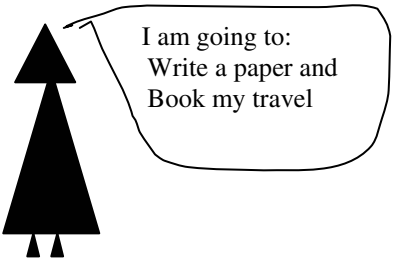
The following examples can help provide a quick and informal introduction to some of the main features of the KGP model.

KGP agents have individual states that are updated as they observe their environment and interact with other agents. They decide dynamically what goals to set themselves depending on their own individual preferences and what they know about their environments.

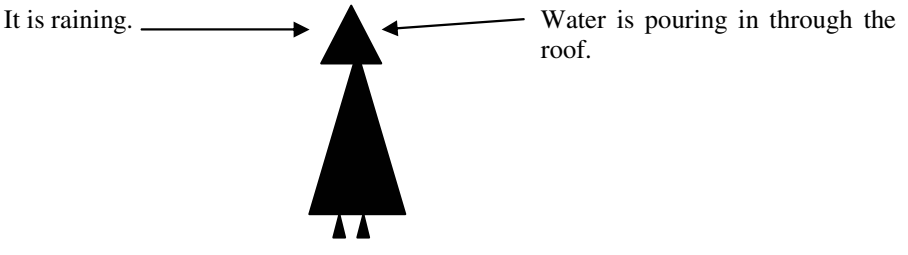
At any particular time the agent may consider a number of potential goals, for example:



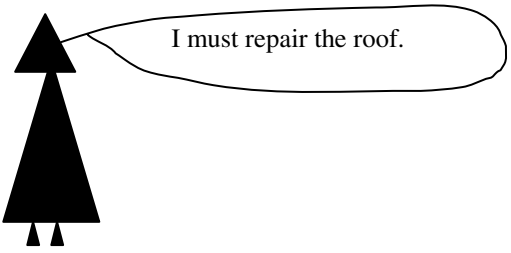
Then depending on its knowledge of the environment, its temporal constraints (e.g. deadline for the paper) and its preferences it can decide which of the possible goals to set itself at that particular time. For example it may decide that the two goals of writing a paper and booking travel should be given highest priority.



It may then proceed with the task of achieving its chosen goals. Concurrently with planning how to achieve its goals and executing actions, the agent observes its environment and records information and communications it receives from the environment and from other agents. For example it may observe that it is raining and that water is pouring in through the roof.



It adapts to changes that it perceives in its environment and circumstances by adjusting or changing its goals, or reacting in some other appropriate way. For example the observation that the roof is leaking may change the agent's priorities and give higher priority to the goal of repairing the roof than the other potential goals.



The agent plans (partially) for its goals and executes actions towards achieving them. For example (informally) the following could be a partial plan for the goal of repairing the roof.

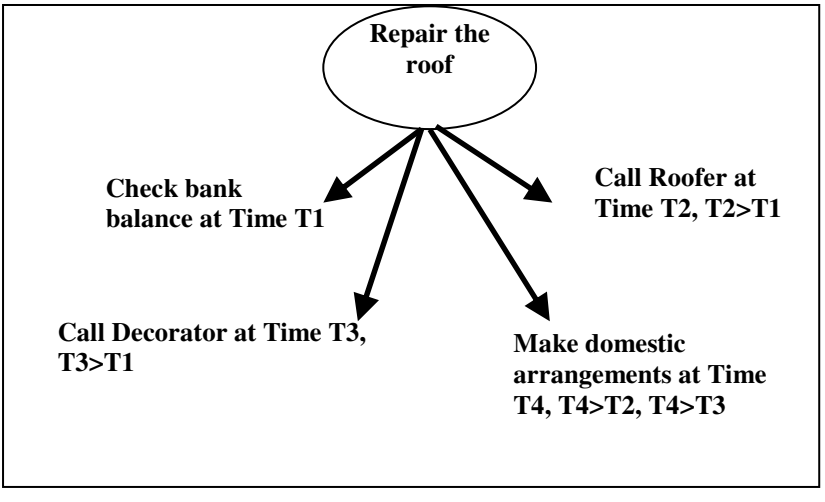
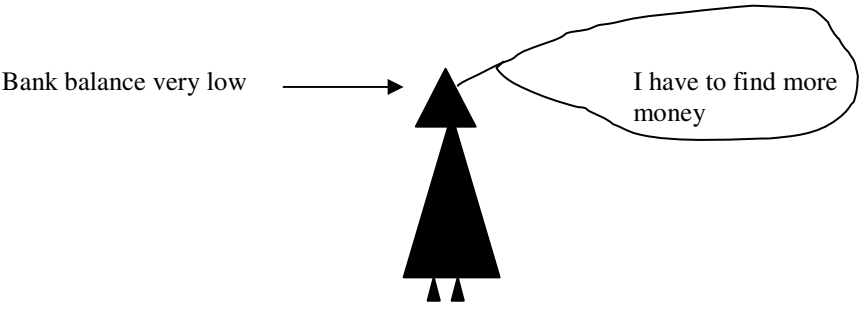


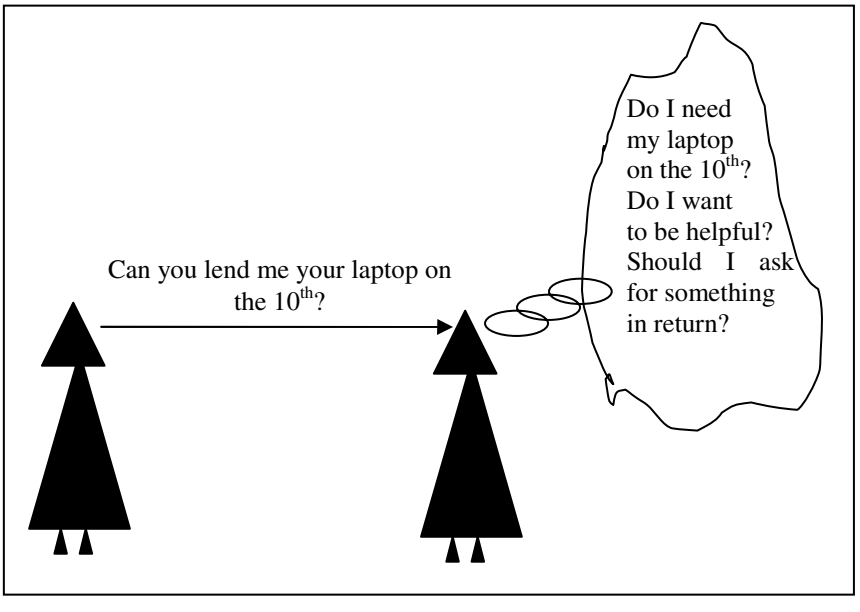
Fig. 1. A partial plan for the goal "Repair the roof"

The partial plan above consists of three actions of checking the bank balance, and calling the roofer and the decorator, and a subgoal of making domestic arrangements which has to be further planned for. All the actions and the subgoal have associated times, possibly as yet undetermined, with some constraints on them, for example that calling the roofer and the decorator should take place after checking the bank balance. Of the three actions here two are *communicative* (calling the roofer and the decorator) and one is *sensing* (checking bank balance).

KGP agents can interleave action execution with planning and observing their environment. Sometimes the result of their action execution or what they observe calls for adjustments to their plans. For example the agent with the plan above may find out that its bank balance is rather low after it executes the action of checking its bank balance. This new knowledge, in turn, can result in the agent setting itself an additional goal of finding more money, and giving this goal appropriate temporal constraints with respect to its other goals.



KGP agents interact with each other. Each one has its own policy on how to respond to messages it receives from others. Such interactions can be used, for example, to ask for resources:



One particularly novel feature of the model is its dynamic context-dependent control. Control is specified by cycle theories that are defined as logic programs with priorities. They allow the agent to determine at run-time what to do next and they allow us to design agents with heterogeneous behaviours.

In the remainder of this paper, we describe the abstract part of the KGP model in more detail, and explain how a designer can proceed to use the model to develop an application. For lack of space our description of the model will not give full details. More details can be found in [9]. Here we summarise the model to the extent of explaining its main features and giving guidelines to the user. Throughout this paper by "user of the model" we mean the person who uses the KGP model to design agents for an application.

2 The KGP Model in a Nutshell

In the KGP model an agent is characterised by the following components:

- An internal mental state, < KB, Goals, Plan >, consisting of a KB which is a collection of knowledge bases, and the agent's (current) goals and plan
- A set of reasoning capabilities
- A sensing capability
- A set of formal state transition rules
- A cycle theory.

The cycle theory orchestrates the application of the transitions, which, in turn, use the capabilities, which use the information in the knowledge bases in the agent's internal mental state. These knowledge bases are updated as the agent receives information from the environment and executes actions in the environment.

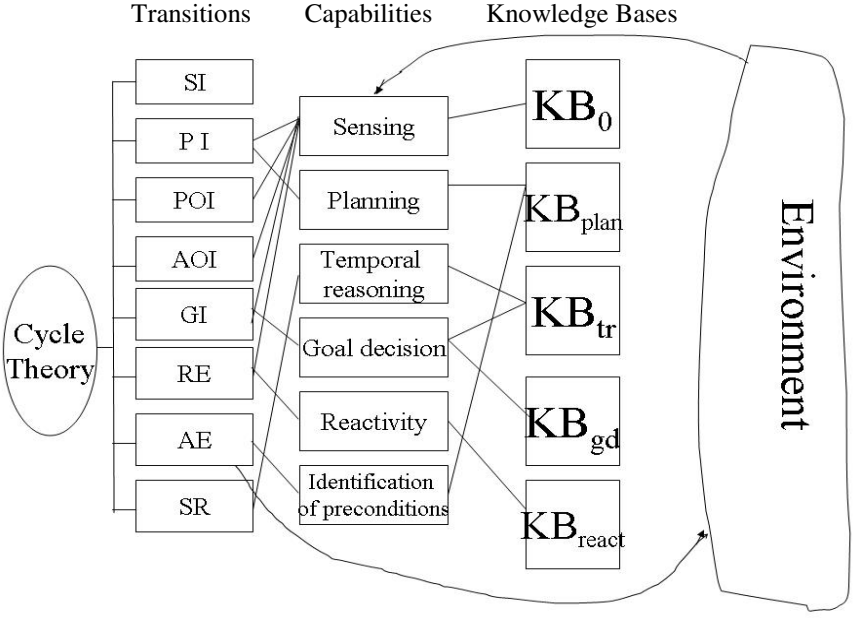


Fig. 2. The architecture of the KGP agent

Some components of the model are fixed, i.e. pre-defined and provided through the implementation platform. These are typically the domain-independent parts of the model:

- The structure of the internal mental state
- The set of capabilities and transitions
- The definition of the capabilities
- The definition of the transitions
- The syntax of the knowledge bases in KB
- Some parts of the knowledge bases

- The syntax of the rules in the cycle theories
- The definition of some of the selection operators (see Section 6) that are used in cycle theories.

Other components of the model are under the control of the application designer using the model. These should be specified by the designer to cater for the requirements and domains of his applications. These components specify domain-dependent knowledge and the specific behaviour requirements of the agent being designed. These user-specified components are:

- Some of the contents of the knowledge bases in KB:
  to cater for knowledge related to a domain or application, knowledge about the priorities of the agents being designed, and the agents' interaction policies.
- The contents of the cycle theories:
  to design the behaviour and profile of agents.
- Some of the definitions of the selection operators:
  to design heuristics affecting the agents' decision making.

Domain specific requirements and heterogeneity are provided by varying the contents of the components that are under the control of the user of the model. We now describe all the components of the model in more detail.

3 The Internal Mental State

The internal mental state of an agent is a triple < KB, Goals, Plan >.

3.1 KB, the Agent's Knowledge Base

KB consists of several modules supporting the reasoning capabilities. These modules are:

- KB0: used to store dynamic data
- KB_{plan}: used for planning
- KB_{tr}: used for temporal reasoning
- KB_{react}: used for reactivity
- KB_{gd}: used for goal decision

Now we describe each of these in some detail.

The first (KB0) is a set of logic facts. The last (KB_{gd}) is a logic program with priorities, and the remaining three (KB_{plan}, KB_{tr}, KB_{react}) are abductive logic programs.

KB0 is a dynamic knowledge base which is *revised* as the agent observes its environment (via its sensing capability) and is *contained* within all the other knowledge bases (and is used by all capabilities).

What the model provides:

KB0 of agent *a* records the following types of information (for details of syntax the reader is referred to [9] or [11]):

- actions which have been executed by *a*, together with the time of the execution *(executed(action, time))*

- actions which have been executed by agents y other than *a*, together with the time of observation by *a (observed(y,action,t))*
- properties (fluent literals) observed by *a*, together with the time of observation *(observed(literal, time)).*

What the user has to provide:
The contents of KB0 are determined by the sensing capability and the Passive and Active Observation transitions (see Section 5), and, of course, by the environment of the agent. The user of the model does not need to design or provide any of the contents of KB0.

KB_{plan} is the knowledge base that is used (in conjunction with KB0) to generate plans for the agents' goals (via the Planning capability and Plan Introduction transition - see Sections 4 and 5). It is an abductive event calculus (AEC) theory. For a description of abductive logic programming see [8], for event calculus see [13], and for abductive event calculus see [14].

What the model provides:
In a nutshell KB_{plan}= < P_{plan}, A_{plan}, I_{plan} >. P_{plan} is a set of rules that define effects and preconditions of actions. In describing the effects of actions it defines a predicate *holds_at(Literal,Time)* in terms of *happens(Action,Time)* and *observed(Literal,Time).*

In particular P_{plan} has two sets of rules, those that are domain-independent and those that are domain-dependent. The domain-dependent part of P_{plan} has to be specified by the user (see later). The following are some of the domain-independent rules in P_{plan}. In these and in the other rules in the remainder of this paper a comma between atoms on either side of the arrow represents the connective "and". All the variables are assumed to be universally quantified over the rule they occur in, unless stated otherwise.

```
holds_at(G,T2)←happens(A,T1), T1 < T2,
          initiates(A,T1,G), not clipped(T1, G,  T2)

holds_at(G,T)←holds_initially(G), 0 < T,
          not clipped(0,G,T)

holds_at(G,T2) ← observed(G,T1), T1 < T2,
           not clipped(T1,G,T2)

clipped(T1,G,T2) ← happens(A,T), terminates(A,T,G),
            T1<T, T<T2

happens(A,T)← executed(A,T)

happens(A,T)← assume_happens(A,T)
```

These rules express that a property G holds at a time if at an earlier time an action initiating it has been executed or assumed (via abduction), or if it held initially (at time 0), or if at an earlier time it has been observed to hold, and, in all cases, provided that G has not been clipped via a terminating action between the two times. The 3rd and 5th rules are bridge rules for connecting the AEC theory to KB0. The 6th rule allows abductions of actions in order to form a plan.

A_{plan}, the set of abducible atoms, consists of *assume_happens(Action,Time).* A plan will contain a set of ground instances of this abducible atom providing the actions of

the plan. A brief example is given below. I_{plan}, the set of integrity constraints. Like P_{plan} it consists of a domain-dependent part and a domain-independent part. The latter consists of the following integrity constraints:

```
holds_at(Literal,Time), holds_at(¬Literal,Time)→ false
assume_happens(Action,Time),precondition(Action,Time,L)
          → holds_at(L, Time)
```

The first constraint expresses that a property and its negation cannot hold at the same time, and the second expresses that if an action is assumed to happen at a time then at that time its precondition must hold.

What the user has to provide:
The user has to provide the domain-dependent parts of P_{plan} and I_{plan}. The domain-dependent part of P_{plan} consists of:

- what holds initially, using the predicate *holds_initially(l)* to denote that a fluent *l* holds initially (at time 0), e.g. *holds_initially(at(john, home))* expresses that John is initially at home,
- what actions initiate and terminate what properties, using the predicates *initiates(a, t, l)* and *terminates(a,t,l)* to denote that action *a*, executed at time *t* initiates or terminates the fluent *l*, respectively, e.g. *initiates(go(X, L1,L2),T,at(X,L2))* and *terminates(go(X, L1,L2),T,at(X,L1))* state that going from location *L1* to *L2* initiates being at *L2* and terminates being at *L1*, and
- the preconditions of actions, using the predicate *precondition(a,t,l)* to denote that fluent *l* is a precondition for executing action *a* at time *t*, e.g. *precondition(go(X, L1,L2),T,at(X,L1))* expresses that a precondition for going from location *L1* to *L2* is being at *L1*.

The domain dependent part of I_{plan} consists of any constraints that are to be specified with respect to the particular agent or environment or application domain. These constraints have to conform to the following syntax:

Conditions → h[t], Tc,

where Conditions is a conjunction of any of the following:

- holds_at (l,t'), where l is a fluent literal and t' is a time variable
- happens (a,t'), where a is an action operator and t' is a time variable
- assume_happens(a,t'), where a and t' are as above,
- temporal constraints,

h[t] is any of the following:

- holds_at (l,t),
- happens (a,t),
- assume_happens(a,t),

and Tc are temporal constraints on t possibly with respect to any time variables in Conditions.

Either of h[t] or Tc may be absent from the head. If both are absent then the head should be *false*.

Examples of such integrity constraints are:

```
assume_happens(go(Person,L,maths_building) ,Time) →
Time>8, Time<23
```

stating that one can go to the maths building only between times 8 and 23.

```
assume_happens(work,Time), assume_happens(rest, Time) →
false
```

stating that the agent cannot work and rest at the same time. As a simple example consider the goal of John being at the maths building at time 10, i.e. holds_at (at(john,maths_building), 10). Given the domain-dependent examples above, a plan for this goal is for John to go to the maths building between the hours of 8 and 10. This plan is denoted as assume_happens(go(john, home, maths_building), T) and T>8 and T<10.

KB_{tr}: In [9] we give a formulation of KB_{tr} that is slightly different from that of KB_{plan}, but here we can assume that KB_{tr} is the same as KB_{plan}. KB_{tr}, the knowledge base for the temporal reasoning part of the model, is used to determine and predict what properties (fluents) hold at given times (via the Temporal Reasoning capability). This functionality is used, for example, when the agent wishes to determine if the preconditions of an action in its plan hold, or to check if (according to what it believes) some of its goals have been achieved.

KB_{react} is used for the reactivity part of the model (Reactivity capability and transition).

What the model provides:
KB_{react} is KB_{plan} with its I_{plan} extended to include *reactive constraints*. The syntax of the reactive constraints is as follows:

Triggers, Conditions → h[t], Tc,

where Conditions, h[t] and Tc are as in the syntax of integrity constraints in I_{plan}, described above, and Triggers is a non-empty conjunction of items of the form *observed(l,t'), observed(c,a,t'), happens(a,t'), assume_happens(a,t'), executed(a,t').*

The intended reading of each reactive constraint is that if the constraint is "triggered" (via matches to *Triggers* found in the agent internal state) and its *Conditions* hold with respect to the internal state, then the constraint "fires", and its conclusion is added to the Goals component of the state if it contains a timed fluent, or to the Plan component if it contains a timed action operator.

What the user has to provide:
The user has to provide all the reactive constraints of KB_{react}. Reactive constraints can be used to represent a number of different things. For example they can be used to represent

- interaction policies,
- condition-action rules, and
- policies for repairing plans.

An example of a reactive constraint representing an interaction policy of agent *a* is:

```
observed(C, tell(C,a,request(R,D,T1)),T),
holds_at(have(R),T1), not holds_at(need(R),T1), T+1<T1
→assume_happens(tell(a,C,accept(request(R,D,T1))),T2),
T2>T, T2<T1
```

This says that if agent a observes that an agent C requests at time T to be given a resource R at a later time T1, and a knows that it has that resource at time T1 and does not need it then a accepts to give C the resource at time T1 and communicates this acceptance to him any time after receiving (observing) the request and before T1. The variable D is an identifier for the dialogue that includes the request and the acceptance of the request.

An example of a reactive rule representing a condition-action rule is:

```
observed(alarm-sound(Room),T), holds_at(in(Room),T)
→assume_happens(leave(Room),T1), T1<T + 2
```

This says that if an alarm sounds in the room you are in leave the room within 2 time points.

An example of a reactive rule representing a specific plan repair policy is:

```
executed(send_message(M),T), observed(network_down, T1),
T1=T+1 → assume_happens(send_message(M), T2), T2>T1+5
```

This says that if you have sent a message and then at the next time point observed that the network is down you should send the message again after waiting at least 5 time units.

Kb_{gd} contains the goal preference policies of the agent. It is used when the agent wishes to decide what goals to set itself (via the Goal Decision capability and transition).

What the model provides:
KB_{gd} has 3 main parts (it also contains KB0):

- the *lower-level part* to generate potential goals,
- the *higher-level part* to specify priorities between the other rules of the theory, effectively allowing to choose amongst the potential goals,
- the *auxiliary part* consisting of rules defining any auxiliary predicates used in the lower and higher level parts.

The syntax for the parts is fixed in the model and is based on logic programming with priorities. We describe the syntax below.

What the user has to provide:
The user has to provide the rules for the 3 parts of KB_{gd} listed above. In doing so the user will determine

- the set of all possible appropriate goals for the agent that is being designed,
- context dependency of potential goals, i.e. rules that determine under what circumstances, depending on temporal constraints, environmental factors and the agent's private knowledge, what goals should be considered, and
- the agent's preferences and priorities, i.e. under what circumstances the agent should commit to which goals.

Note that the possible appropriate goals for the agent should guide the user towards what needs to be specified in KB_{plan}, i.e. it would make sense for KB_{plan} to provide specification of actions (through the *initiates, terminates* and *precondition* predicates) that can help towards achieving some or all of these goals. In other words it would be

appropriate to incorporate in the model the knowledge that can potentially be used to generate plans for the potential goals of the agent.

The lower-level part of KB_{gd} consists of rules of the form

name of the rule: $G[t], Tg \leftarrow L_1, \ldots, L_n, Tc$ (n>0 or n=0)

where

- the L_i are either time dependent conditions of the form *holds_at(l,t)*, or time dependent conditions formulated in terms of auxiliary predicates defined in the auxiliary part of KB_{gd},
- *G* is a goal fluent (see Section3.2 and the examples below) chosen by the user,
- *Tg* is a (possibly empty) set of temporal constraints,
- *t* is a time variable, assumed to be existentially quantified with the scope the head of the rule,
- *Tc* are temporal constraints on the time variables in the body of the rule.

All variables, except t, are implicitly universally quantified over the rule. Each rule in the lower-level part is given a name. Examples of lower-level rules are:

```
gd(dinner): make_dinner(T) ← holds_at(finished_work,T)

gd(repair): repair_roof(T) ← holds_at(leaky_roof,T)
```

These state that making dinner is a potential goal when work is finished and repairing the roof is a potential goal when the roof is leaking.

The higher-level part of KB_{gd} consists of rules of the form

name of the rule: $h\_p(rule1, rule2) \leftarrow L_1, \ldots, L_n, Tc$ (n>0 or n=0)

where

- the L_i are *Tc* are as described as in the lower-level part, and
- *rule1* and *rule2* are names of other rules in KB_{gd}.

These higher-level rules represent priorities amongst rules in the lower-level part or other priority rules in the higher-level part. Each rule in the higher-level part is given a name. Examples of higher-level rules are:

```
gd_pref(X,Y):h_p(gd(X), gd(Y)) ← type(X,TX),
                type(Y,TY), more_urgent_wrt_type(TX,TY)
```

This states that the rule called gd(X) should be given higher priority than the rule called gd(Y) whenever X is a more urgent type of goal compared to Y.

The auxiliary part is simply a logic program defining any auxiliary predicates occurring in the other parts. In addition, it can contain statements of incompatibility using the predicate *incompatible(g1,g2)* denoting that two goals *g1* and *g2* are incompatible (to hold at the same time). Examples of the auxiliary part rules are:

```
type(dinner, optional)

type(repair, required)

more_urgent_wrt_type(required,optional)

incompatible(make_dinner, repair_roof)
```

These collection of example rules for the 3 parts of KB_{gd} ensure that whenever both making dinner and repairing the roof are potential goals the latter will be chosen as the one with higher priority.

3.2 Goals and Plan

What the model provides:
The representation of a goal in the state is a timed fluent l[t], for example *has_driving_licence(john, T1)*, where T1 may be constrained in the state, for example by the temporal constraints *10<T1, T1<20*. There are two types of goals:

- Mental (under the control of the agent), e.g. *be_at_the_airport(T), T<18*
- Sensing (not under the control of the agent and observable by sensing the external environment), e.g. *request_accepted(T), raining(T).*

When a goal l[t] in the state is selected for planning it is automatically represented as holds_at(l,t).

The representation of a Plan in the state is a set of partially ordered actions. An action is a timed operator a[t], e.g *pay_fine(john, T)*, where T may be constrained in the state, for example by the temporal constraint T1<T, T<T3. There are three types of actions:

- Physical e.g. *do(clear_table, T)*
- Sensing e.g. *sense(connection_on, T)*
- Communicative e.g. *tell(x, y, request(r1, d, T),T1)*

All the time variables associated with goals and actions are assumed to be existentially quantified over the whole state. Goals and actions can be viewed as organised in a tree structure, showing associations of goals/subgoals/actions for ease of revision and partial planning.

Below is an (informal) example of goals/actions tree in state of an agent called *a*.

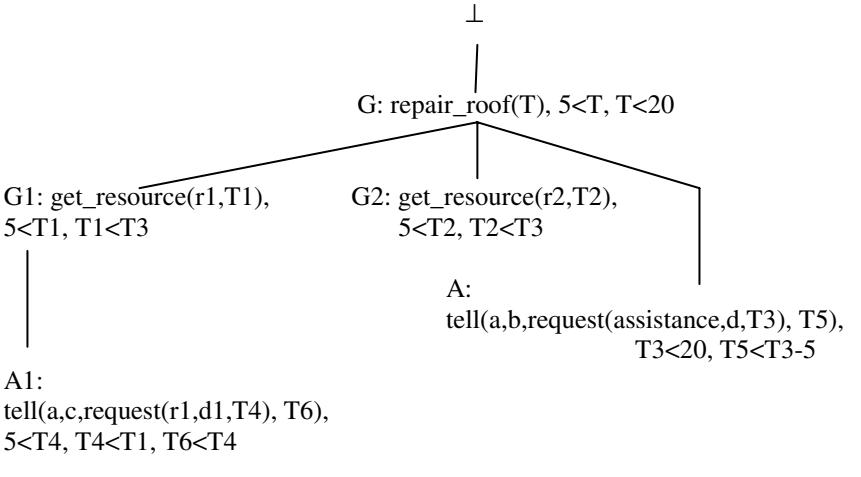


Fig. 3. A Goals/Actions Tree

In this tree the root is represented by the symbol ⊥. The *top level* goal is to repair the roof at a time between times 5 and 20. A partial plan for this goal consists of the two subgoals G1 and G2 of getting two resources r1 and r2 within the specified temporal constraints, and an action A of requesting assistance from agent b. A (full) plan for goal G1 consists of action A1 of requesting the resource from agent c, with the specified temporal constraints.

What the user has to provide:
For the Goals and Plan components of the agent's internal state the user does not need to provide anything. The goals in Goals will result from the information the user provides in KB_{gd}, KB_{plan} and KB_{react}. The actions in Plan will result from the information the user provides in KB_{plan} and KB_{react}.

4 The Capabilities

As mentioned in Section 2 the model provides a sensing capability and a number of reasoning capabilities. The model provides all the necessary specifications for these. Below we summarise these capabilities.

The Sensing Capability: This allows the agent to observe the environment and to receive messages from other agents. The agent observes actions executed by other agents and fluents holding in the environment. These observations are made either passively via the Passive Observation Introduction transition (see Section 5) or actively by the agent seeking specific information, via the Active Observation Introduction transition (see Section 5). The results of the observations are recorded in KB0, as described in Section 3.

The Reasoning Capabilities: There are 5 reasoning capabilities:

1. Planning: generates partial plans for given sets of goals in the internal state of the agent
2. Temporal reasoning: makes predictions about properties holding in the environment
3. Reactivity: reacts to perceived changes in the environment by generating goals and actions to be added to the internal state of the agent
4. Identification of preconditions of actions: identifies the preconditions of given sets of actions
5. Goal decision: determines how the top level goals of the agent (and consequently all the goals of Goals in the internal state of the agent) should be revised to take into account the agent's preferences and the perceived changes in the environment.

All the reasoning capabilities are formally specified in the model, the first 4 using abductive logic programming, and the last using logic programming with priorities. The formal specifications can be found in [9]. To give a flavour we give a simplified specification of the Planning capability.

Specification of the Planning capability:
KB, Goals, Plan, $(G_1, \ldots, G_n) \models^{\Gamma}_{plan} (PP_1, \ldots, PP_n)$ such that

- $P_{plan} \cup Goals \setminus G_i \cup Plan \cup PP_i \models_{LP} G_i$, for each i from 1 to n
- $P_{plan} \cup Goals \cup Plan \cup PP_1 \cup \ldots \cup PP_n \models_{LP} I_{plan}$

- There is a substitution σ for all the time variables in Goals, Plan, G_1, …, G_n, PP_1,…, PP_n, that satisfies all temporal constraints in Goals, Plan, G_1, …, G_n, PP_1,…, PP_n and allows all time variables of any actions in the PP_i to be instantiated by times in the future of Γ.

Here $\models_{plan}$ denotes the Planning capability and $\models_{LP}$ denotes any semantics for logic programming. The specification above states that the Planning capability takes as input

- the agent internal state
- a set G_1, …, G_n of goals (to be planned for), which would be a subset of the goals in Goals
- a time Γ (the time the capability is called),

and produces as output a partial plan PP_i for each input goal G_i, such that

- each PP_i entails its associated goal G_i in the context of the state (without the G_i),
- all the partial plans together with the internal state entail all the integrity constraints in I_{plan},
- all the resulting temporal constraints, including any new ones generated (and any new instantiations of time parameters) are satisfiable together, and
- the temporal constraints of the new planned actions allow the actions to be performed in the future of Γ.

In a nutshell the Planning capability generates consistent, feasible partial plans for all the input goals. We gloss over exactly what a partial plan is. Examples have been given in Sections 1 and 3, and details are available in [9,14].

Notice that this specification is parametric on:

- $\models_{LP}$, i.e. some semantics for logic programs, and
- some semantics underlying constraint satisfaction.

In addition the formal specification of the Goal decision capability is parametric on some semantics $\models_{PR}$ for logic programs with priorities. The computational model of the KGP commits to concrete instances of the above: 3-valued completion semantics for $\models_{LP}$ and argumentation based semantics for a concrete framework, LPwNF [3] of $\models_{PR}$.

For the rest of the capabilities we give summary, informal specifications:

- The Temporal reasoning capability takes as input KB_{tr} and a timed fluent and determines if the fluent holds at the specified time.
- The Reactivity capability takes as input the internal state of the agent and a time of application of the capability, and returns as output all the "reactions" that are "fired" at that time from KB_{react}.
- The Identification of preconditions capability takes as input KB_{plan} and a set of timed action operators and returns the preconditions of those actions as determined by KB_{plan}.
- The Goal decision capability takes as input KB_{gd} and a time of application of the capability, and returns all the goals that are determined by KB_{gd} to have highest priority at that time.

5 The Transitions

Transitions use the capabilities and change the internal state of the agent. The model provides all the necessary specifications for the transitions. There are 8 transitions. They are:

1. Goal Introduction - GI: It replaces Goals in the state by the highest priority goals that the Goal decision capability generates.
2. Plan Introduction - PI: It uses the Planning capability and extends the state with the resulting partial plans for a selected set of goals.
3. Reactivity - RE: It extends the Goals and/or Plan components of the state with the reactions (goals and/or actions) that the Reactivity capability generates.
4. Action Execution - AE: It executes (a selected set of) actions and records their execution in KB0. It uses the Sensing capability for the execution of sensing actions.
5. Passive Observation Introduction - POI: It records in KB0 any (unsolicited) information observed in the environment or communication received from other agents. It uses the Sensing capability.
6. Active Observation Introduction - AOI: It senses the environment for a specific set of properties (fluents) and records the result in KB0. It uses the Sensing capability.
7. Sensing Introduction - SI: It adds new sensing actions to the Plan for sensing the environment to determine whether or not preconditions of some existing actions in Plan hold.
8. State Revision - SR: It revises the Goals and Plan components of the state by removing goals that are achieved or timed-out and their children, and actions that have been executed or timed-out. It uses the Temporal reasoning capability.

The transitions are specified in the following general form:

$$T: \quad \frac{S=\langle KB, Goals, Plan\rangle \text{ , Input} \quad \text{at a time } \tau}{S' =\langle KB', Goals', Plan'\rangle}$$

denoting that the transition T takes a state S and an input at a time τ, and changes the state to S'. The Input may be missing from the specification of some of the transitions.

Transitions typically:

- call some capabilities and/or check for temporal constraint satisfaction, and
- have an input computed by *selection operators* (see Section 6).

The input is either a set of actions (to be executed in AE, for example), or a set of goals (to be planned for in PI), or a set of fluents (to be sensed in the environment in AOI, for example).

As an example we give the specification of the Plan Introduction (PI) transition below. Many details are glossed over, for lack of space.

$$PI: \quad \frac{S=\langle KB, Goals, Plan\rangle, SGs \quad \tau}{S'=\langle KB, Goals', Plan'\rangle}$$

where S' is determined as follows:

- The planning capability $\models_{plan}$ is utilised at time τ with input the set of goals SGs.
- $\models_{plan}$ will return partial plans for each goal G in SGs, the partial plans consisting of (sub)goals, actions and temporal constraints.
- The returned (sub)goals and actions are added to Goals and Plan, respectively, together with their temporal constraints.

6 Cycle Theories for Declarative Control

In the KGP model the agent control of the operations, i.e. the orchestration of the transitions, is via cycle theories. This is quite different compared with some other agent systems where a conventional control mechanism dictates a fixed sequence of operations. The KGP cycle theories determine the sequences of transitions *dynamically* and *declaratively*, providing flexible control that can be designed to capture specific agent behaviour profiles and to fit specific environments or applications.

The cycle theories are specified using logic programs with priorities. They are described in detail in [9], and in summary in [11]. Some behaviour profiles resulting from varying cycle theories are described in [15] and [1]. Here we give a summary with more emphasis on distinguishing what the model provides and what the user needs to add.

What the model provides:
A cycle theory is a (meta-)logic program with priorities T_{cycle} to reason about which transition should be chosen when. It consists of:

- a *basic* part T_{basic} to reason about which transition could be next (in some given state and at a given time), initially or after a transition that has just been executed,
- a *behaviour* part $T_{behaviour}$ to decide which transition (amongst the possibly many potential ones) will be next, and
- an auxiliary part, providing definitions of auxiliary predicates used in the other two parts.

T_{basic} consists of rules of the form:

$$r_{T1|T2}(S',X') : T2(S',X') \leftarrow T1(S,X,S'), EC(S', \tau, X'), time\_now(\tau)$$

where S, S' are states, T1, T2 are transition names (PI, GI, etc), X' is input to T2, and EC is a (possibly empty) conjunction of *enabling conditions* (defined in terms of the *core selection operators* described below).

The rule states that after transaction T1 has been performed with some input X and changing the state from S to S', then transition T2 is a possible follow-up provided at the current time the enabling conditions EC hold and produce an input X' for transition T2. The rule is given the name $r_{T1|T2}(S',X')$. Notice the predicative representation of transitions in cycle theory rules. A transition represented as

$$T: \quad \frac{S=<KB, Goals, Plan> , Input \quad \text{at a time } \tau}{S'=<KB',Goals',Plan'>}$$

as seen in Section 5, is represented as an atom in the predicate T:

T(S, Input, S',τ)

and sometimes, for brevity, with some parameters omitted.

$T_{behaviour}$ consists of rules of the form:

$$R^{T}_{N1|N2}: \; r_{T|N1}(S,X1) > r_{T|N2}(S,X2) \leftarrow BC(S,X1,X2,\tau), \; time\_now(\tau)$$

where S is a state, N1, N2, T are transition names, X1 is input to N1, X2 is input to N2, BC is a (possibly empty) conjunction of *behaviour conditions* (defined in terms of the *heuristic selection operators* described below).

The rule states that after transition T, transition N1 is preferred to N2 when the behaviour conditions hold at the current time τ and produce inputs X1 and X2, respectively for N1 and N2. The behaviour rule is given the name $R^{T}_{N1|N2}$.

The auxiliary part of T_{cycle} consists of the definitions of any predicates occurring in the enabling and behaviour conditions, and rules of the form *incompatible(T(S,X), T'(S',X'))* stating that different transitions are incompatible with each other as are different calls to the same transition with different inputs (to be executed at the same time).

The enabling conditions of the rules in T_{basic} are defined in terms of the *core selection operators*. These selection operators compute the inputs to the transitions and help cycle theories to determine the next possible transition. There are 4 core selection operators:

- Action selection - $c_{AS}(S, \tau)$: selects a set of actions in the current state for execution.
- Goal selection - $c_{GS}(S, \tau)$: selects a set of goals in the current state to be planned for.
- Fluent selection - $c_{FS}(S, \tau)$: selects a set of fluents to be sensed in the environment.
- Precondition selection - $c_{PS}(S, \tau)$: selects a set of action preconditions to be sensed.

The definitions of these operators are given within the model. For example $c_{GS}(S, \tau)$ is the set of all goals in the state S at time τ which have not been achieved yet, are not timed out and are not the children of goals that have been achieved or are timed out. Analogous to the core selection operators there are 4 *heuristic selection operators* which are used to define the behaviour conditions in $T_{behaviour}$. The definitions of these are under the control of the user.

Given an agent's cycle theory T_{cycle}, the agent's behaviour is characterised as a (possibly infinite) sequence of transitions

$$T_1(S_0,X_1,S_1, \tau_1),\ldots,T_i(S_{i-1},X_i,S_i, \tau_i), \; T_{i+1}(S_i,X_{i+1},S_{i+1}, \tau_{i+1}),\ldots.$$

such that

- S_0 is some initial state for the agent
- τ_i is given by some internal clock
- T_{cycle}, $T_i(S_{i-1},X_i, S_i, \tau_i)$, time_now(τ) $\models_{pr} T_{i+1}(S_i,X_{i+1}, S_{i+1}, \tau_{i+1})$.

$\models_{pr}$ denotes some semantics for logic programs with priorities. The abstract KGP model is parametric with respect to this. The computational model chooses argumentation based semantics for a concrete framework of $\models_{pr}$ [3].

A complete specification of a cycle theory, called the *normal cycle theory*, can be found in [15]. We cannot reproduce it here for lack of space.
What the user has to provide:

The user can provide his own rules for all the 3 components of T_{cycle} conforming to the general syntax. The following are some examples.

Examples of T_{basic}:

$$r_{PI|AE}(S',As): AE(S',As) \leftarrow PI(S,Gs,S'), As=c_{AS}(S', \tau), As \neq \{\}, time\_now(\tau)$$

This states that a Plan Introduction transition may be followed by an Action Execution transition, if there are actions to be executed (identified by the core selection operator for action selection c_{AS}).

$$r_{POI|RE}(S',\_): RE(S',\_) \leftarrow POI(S,\_,S')$$

This states that a Passive Observation Introduction transition may be followed by a Reactivity transition, unconditionally. Note that the Reactivity transition requires no input computed by any of the selection operators.

Examples of $T_{behaviour}$:

$$R^{PI}_{AE|N}: r_{PI|AE}(S,As) > r_{PI|N}(S,X) \leftarrow not\ unreliable\_pre(As)$$

for all transitions $N \neq AE$.

$$R^{PI}_{SI|AE}: r_{PI|SI}(S,Ps) > r_{PI|AE}(S,As)) \leftarrow unreliable\_pre(As)$$

These two rules state that after Plan Introduction, the transition Action Execution is preferred to any other, unless there are actions amongst the actions selected for execution whose preconditions are "unreliable" and need checking, in which case Sensing Introduction will be given preference. The predicate *unreliable_pre* has to be defined in the auxiliary part of T_{cycle}.

By varying the rules of the cycle theory the behaviour of the agent can be varied. Two different profiles of behaviour, called *focussed* and *careful*, obtained in this way are described in [15], where cycle theories are provided for each profile. With the focussed profile an agent concentrates on one goal at a time until it achieves it or it is convinced that it is unachievable. With the careful profile, after any transition the agent revises its state via the SR transition to ensure that unachievable or unnecessary goals and actions are revised away as soon as possible. A collection of other profiles, their properties and their associated cycle theories have been proposed in [1].

7 Conclusion

In this paper we have provided a tutorial on the KGP model of agency, concentrating on the abstract counterpart of the model. The tutorial has aimed to provide an

overview of the model and give some user guidance. For each module of the abstract model we have summarised the domain-independent part which is provided by the model, and available through the implementation platform. In addition, for each module we have discussed the features of the domain-dependent part which the user has to provide in order to specify the particular requirements of the application.

This tutorial should help the user make a start on designing an agent in the KGP model. On its own, however, it is not sufficient for providing guidance up to and including the implementation stage. Further guidance on implementation is needed. This will become available when the platform becomes publicly accessible.

Acknowledgements

I am grateful to the anonymous reviewers for their helpful comments on an earlier draft of this paper. Work on the KGP model was funded by the IST programme of the EC, FET under the IST-2001-32530 SOCS project, within the GC proactive initiative.

References

1. M. Alberti, F. Athienitou, A. Bracciali, F. Chesani, U. Endriss, M. Gavanelli, A. Kakas, E. Lamma, W. Lu, P. Mancarella, P. Mello, F. Sadri, K. Stathis, F. Toni, P. Torroni: Verifiable Properties of Societies of Computees, Technical report, SOCS Consortium, Deliverable D13, U. Endriss, F. Sadri (eds.), will be available at http://lia.deis.unibo.it/research/socs/guests/publications/ (2005)
2. A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, F. Toni: The KGP Model of Agency for Global Computing: Computational Model and Prototype Implementation, Global Computing 2004 Workshop, Springer Verlag LNCS 3267 (2005) p. 342
3. Y. Dimopoulos, A.C. Kakas: Logic Programming Without Negation as Failure, in Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon (1995) p. 369
4. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, F. Toni: The CIFF Proof Procedure for Abductive Logic Programming With Constraints, JELIA'2004, International Conference on Logics in AI, Lisbon, Portugal, September 2004, Springer Verlag LNAI 3229 (2004) p. 31
5. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, F. Toni: Abductive Logic Programming with CIFF: System Description, JELIA'2004, International Conference on Logics in AI, Lisbon, Portugal, September 2004, Springer Verlag LNAI 3229 (2004) p. 680
6. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, F. Toni: Abductive Logic Programming with CIFF: Implementation and Applications, CILC04, Convegno Italiano di Logica Computazionale, 16-17 June 2004, Parma, Italy, Research Report Quaderno del Dipartimento di Matematica, Universita' di Parma, n. 390 (2004) p. 28
7. Gorgias: Argumentation and Abduction (http://www.cs.ucy.ac.cy/~nkd/gorgias)
8. A.C.Kakas, R.A. Kowalski, F. Toni: The Role of Abduction in Logic Programming, in Handbook of Logic in Artificial Intelligence and Logic Programming, D.M. Gabbay, C.J. Hogger, J.A. Robinson (eds.), volume 5, Oxford University Press (1998) p.235
9. A.C. Kakas, E. Lamma, P.Mancarella, P. Mello, K.Stathis, and F.Toni: Computational Model for Computees and Society of Computees, Technical report, SOCS Consortium, Deliverable D8, will be available at http://lia.deis.unibo.it/research/socs/guests/publications/ (2003)

10. A.C. Kakas, P.Mancarella, F. Sadri, K.Stathis, and F.Toni: A Logic-based Approach to Model Computees, Technical report, SOCS Consortium, Deliverable D4, will be available at http://lia.deis.unibo.it/research/socs/guests/publications/ (2003)
11. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, F. Toni: The KGP Model of Agency, ECAI04, General European Conference on Artificial Intelligence, August 23-27, Valencia, Spain (2004) p. 33
12. A.C. Kakas, P.Mancarella, F.Sadri, K.Stathis, and F.Toni: Declarative Agent Control, 5th Workshop on Computational Logic in Multi-Agent Systems (CLIMA V), 29-30 September, J.Leite and P.Torroni (eds.) (2004) p. 212
13. R.A. Kowalski, M. Sergot: A Logic-based Calculus of Events, New Generation Computing, 4(1):67-95 (1986)
14. P.Mancarella, F.Sadri, G.Terreni, and F.Toni: Planning Partially for Situated Agents, 5th Workshop on Computational Logic in Multi-Agent Systems (CLIMA V), 29-30 September 2004, J.Leite and P.Torroni (eds.)
15. F. Sadri and F. Toni: Variety of behaviours Through Profiles in Logic-based Agents, in this volume
16. F. Sadri, K. Stathis, F. Toni: Normative KGP Agents: A Preliminary Report, Proc. NorMAS2005, 1st International Symposium on Normative Multi-Agent Systems, AISB convention (2005)
17. F. Sadri, K. Stathis, F. Toni: Normative KGP Agents, Computational and Mathematical Organization Theory (2006) (to appear)
18. Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali: PROSOCS: a Platform for Programming Software Agents in Computational Logic, in J. Müller and P. Petta (eds.), Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation" (AT2AI-4 - EMCSR'2004 Session M), Vienna, Austria, 13-16 April (2004) p. 523
19. JXTA: http://www.jxta.org