Development of Intelligent Systems and Multi-Agents Systems with Amine Platform

Adil KABBAJ
INSEA
Rabat, Morocco, B. P 6217
akabbaj@insea.ac.ma

http://sourceforge.net/projects/amine-platform

Abstract. Amine is a Java open source multi-layer platform dedicated to the development of intelligent systems and multi-agents systems. This paper and companion papers [2, 3] provide an overview of Amine platform and illustrate its use in the development of dynamic programming applications, natural language processing applications, multi-agents systems and ontology-based applications.

1. Introduction

Amine is a Java open source multi-layer platform and a modular Integrated Development Environment, dedicated to the development of intelligent systems and multi-agents systems. Amine is a synthesis of 20 years of works, by the author, on the development of tools for various aspects of Conceptual Graph theory.

This paper shows how dynamic programming, natural language processing and multi-agents systems can be developed using Amine. The companion paper [2] illustrates the use of Amine in the development of ontology based applications. A forthcoming paper will describe the use of Amine in problem-solving applications; and especially in the development of a strategic card game called Tijari (which has some similarity with Bridge card game).

We urge the reader to consult Amine web site[1] for more detail. Source code, samples and documentation can be downloaded from sourceforge site[2].

The paper is organized as follows: section 2 introduces briefly Amine platform. A more detailed description of Amine architecture is provided in the companion paper [3]. Section 3 introduces the re-engineering, extension and integration of Prolog+CG language [4, 5] in Amine. Section 4 introduces the re-engineering, extension and integration of Synergy language [6, 7] in Amine. It shows also how Synergy has been extended to enable dynamic programming. Section 5 discusses the use of the new version of Prolog+CG in the development of natural language processing applications. Section 6 illustrates briefly the use of Amine, in conjunction with Jade[3]

1 amine-platform.sourceforge.net
2 sourceforge.net/projects/amine-platform
3 jade.tilab.com/

(Java Agent Development Environment), in the development of a multi-agents system called Renaldo. A forthcoming paper will describe the development of Renaldo in more detail.

Section 7 provides a comparison of Amine with other CG tools. Section 8 outlines some current and future work. Section 9 concludes the paper.

2. An overview of Amine Platform

Amine is a modular integrated environment composed of four hierarchical layers: a) *ontology layer* provides "structures, processes and graphical interfaces" to specify the "conceptual vocabulary" and the semantic of a domain, b) *algebraic layer* is build on top of the ontology layer: it provides "structures, operations and graphical interfaces" to define and use "conceptual" structures and operations, c) *programming layer* is build on top of the algebraic layer: it provides "programming paradigms/languages" to define and execute "conceptual" processes and, d) *multi-agent layer* provides plugs-in to agent development tools, allowing for the development of multi-agent systems.

More specifically:

1. *Ontology layer:* It concerns the creation, edition and manipulation of *multi-lingua ontology*. The companion paper [2] presents this layer in detail (including ontology meta-model and ontology related processes).
2. *Algebraic layer:* this layer provides several types of structures and operations: elementary data types (AmineInteger, AmineDouble, String, Boolean, etc.) and structured types (AmineSet, AmineList, Term, Concept, Relation and Conceptual Graph). In addition to operations that are specific to each kind of structure, Amine provides a set of basic common operations (clear, clone, toString, etc.) and various common matching-based operations (match, equal, unify, subsume, maximalJoin and generalize). Structures can be generic; they can contain variables and the associated operations take into account *variable binding* (the association of a value to a variable) and *binding context* (the programming context that determines how variable binding should be interpreted and resolved, i.e. how to associate a value to a variable and how to get the value of a variable).

   Amine structures and operations (including CG structure and CG operations) are APIs that can be used by any Java application. They are also "inherited" by the higher layers of Amine.

   The companion paper [3] provides more detail on the algebraic layer of Amine. It highlights also the use of Java interfaces to enhance the genericity of Amine. See also Amine web site for mode detail on this basic feature of Amine.
3. *Programming layer:* Three complementary programming paradigms are provided by Amine: a) *pattern-matching and rule-based programming paradigm,* embedded in PROLOG+CG language which is an object based and CG-based extension of PROLOG language, b) *activation and propagation-based programming paradigm*, embedded in SYNERGY language, and c) *ontology or memory-based programming paradigm* which is concerned by incremental and automatic integration of knowledge in an ontology (considered as an agent

memory) and by information retrieval, classification and other related ontology/memory-based processes.

4. *Agents and Multi-Agents Systems layer:* Amine can be used in conjunction with a Java Agent Development Environment to develop multi-agents systems. Amine does not provide the basic level for the development of multi-agents systems (i.e. implementation of agents and their communication capabilities using network programming) since this level is already offered by other open source projects (like Jade). Amine provides rather plugs-in that enable its use with these projects in order to develop multi-agents systems.

Amine provides also several graphical user interfaces (GUIs): Ontology GUI, CG Notations editors GUI, CG Operations GUI, Dynamic Ontology GUI, Ontology processes GUI, Prolog+CG GUI and Synergy GUI. *Amine Suite Panel* provides an access to all GUIs of Amine, as well as an access to some ontology examples and to several tests that illustrate the use of Amine structures and their APIs. Amine has also a web site, with samples and a growing documentation.

Amine four layers form a *hierarchy*: each layer is built on top of and use the lower layers (i.e. the programming layer inherits the ontology and the algebraic layers). However, a lower layer can be used by itself without the higher layers: the ontology layer (i.e. with the associated APIs) can be used directly in any Java application without the other layers. Algebraic layer (i.e. with the associated APIs) can be used directly too, etc. Among the goals (and constraints) that have influenced the design and implementation of Amine was the goal to achieve a higher level of modularity and independence between Amine components/layers.

Amine platform can be used as a modular integrated environment for the development of intelligent systems. It can be used also as the kernel; the basic architecture of an *intelligent agent*: a) the ontology layer can be used to implement the dynamic memory of the agent (agent's ontology is just a perspective on the agent's memory), b) the algebraic layer, with its various structures and operations, can be used as the "knowledge representation capability" of the agent, c) the programming layer (i.e. dynamic ontology engine, Prolog+CG and Synergy) can be used for the formulation and development of many inference strategies (induction, deduction, abduction, analogy) and cognitive processes (reasoning, problem solving, planning, natural language processing, dynamic memory, learning, etc.), d) Synergy language can be used to implement the reactive and event-driven behaviour of the agent.

One long-term goal of the author is to use Amine, in conjunction with Java Agent Development Environments (like Jade), to build various kinds of intelligent agents, with multi-strategy learning, inferences and other cognitive capabilities.

The group of Peter Ohrström and Henrik Scharfe has developed on-line course that covers some parts of Amine Platform[4]. Amine is used by the author to teach Artificial Intelligence (AI) courses. Amine is suited for the development of projects in various

[4] www.huminf.aau.dk/cg/

domains of AI (i.e. natural language processing, problem solving, planning, reasoning, case-based systems, learning, multi-agents systems, etc.).

3. Re-engineering, extension and integration of Prolog+CG in Amine

Prolog+CG has been developed by the author as a “stand-alone” programming language [4, 5]. The group of Peter Ohrström developed a very good on-line course on some aspects of Prolog+CG. Let us recall three key features of previous versions of Prolog+CG:

- CG (simple and compound CGs) is a basic and primitive structure in Prolog+CG, like list and term. And like a term, a CG can be used as a structure and/or as a representation of a goal. Unification operation of Prolog has been extended to include CG unification. CG matching-based operations are provided as primitive operations.
- By a supplementary indexation mechanism of rules, Prolog+CG offers an object based extension of Prolog.
- Prolog+CG provides an interface with Java: Java objects can be created and methods can be called from a Prolog+CG program. Also, Prolog+CG can be activated from Java classes.

The interpreter of Prolog+CG, that takes into account these features (and others) has been developed and implemented in Java by the author.

The above three key features are still present in the new version of Prolog+CG but the re-engineering of Prolog+CG, which was necessary for its integration in Amine platform, involved many changes in the language (and its interpreter). Five main changes are of interest (see Amine Web Site for more details):

- Type hierarchy and Conceptual Structures (CSs) are no more described in a Prolog+CG program. Prolog+CG programs are now interpreted according to a specified ontology that includes type hierarchy and CSs. Also, a Prolog+CG program has the current ontology as support: Prolog+CG interpreter attempts first to interpret each identifier in a program according to the current lexicon of the current ontology. If no such identifier is found, then the identifier is considered as a simple identifier (without any underlying semantic).
- The notion of project is introduced: user can consult several programs (not only one) that share the same ontology.
- Prolog+CG inherit the first two layers of Amine: all Amine structures and operations are also Prolog+CG structures and operations. And of course, Prolog+CG user can manipulate the current ontology and the associated lexicons according to their APIs.
- The interface between the new version of Prolog+CG and Java is simpler and “natural” in comparison with previous interfaces (see Amine Web site for more detail).
- Interoperability between Amine components: Prolog+CG can be used in conjunction with the other components of Amine (i.e. dynamic ontology engine and Synergy can be called/used from a Prolog+CG program).

2. Re-engineering, extension and integration of Synergy in Amine

In [6, 7] we proposed *CG activation-based mechanism* as a computation model for *executable conceptual graphs*. Activation-based computation is an approach used in visual programming, simulation and system analysis where graphs are used to describe and simulate sequential and/or parallel tasks of different kinds: functional, procedural, process, event-driven, logical and object oriented tasks. Activation-based interpretation of CG is based on *concept lifecycle*, *relation propagation rules* and *referent/designator instantiation*. A concept has a state (which replaces and extends the notion of control mark used by Sowa) and the concept lifecycle is defined on the possible states of a concept. Concept lifecycle is similar to process lifecycle (in process programming) and to active-object lifecycle (in concurrent object oriented programming), while relation propagation rules are similar to propagation or firing rules of procedural graphs, dataflow graphs and Petri Nets.

SYNERGY is a visual multi-paradigm programming language based on CG activation mechanism. It integrates functional, procedural, process, reactive, object-oriented and concurrent object-oriented paradigms. The integration of these paradigms is done using CG as the basis knowledge structure, without actors or other external notation. Previous versions of Synergy have been presented [6, 7]. The integration of Synergy in Amine required re-engineering work and some changes and extensions to the language and to its interpreter. New features of Synergy include:

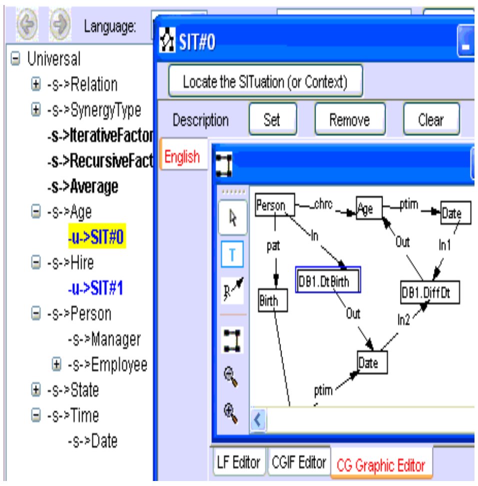
- Long-term memory introduced in previous definitions of Synergy corresponds now to ontology that plays the role of a support to a Synergy “expression/ program”,
- Previous versions of Synergy did not have an interface with Java. The new version of Synergy includes such an interface; Java objects can be created and methods activated from Synergy. This is an important feature since user is not restricted to (re)write and to define anything in CGs. Also, primitive operations are no more restricted to a fixed set of operations.
- The new version of Synergy has an access to the two first layers of Amine. Also, since Prolog+CG, Synergy and dynamic ontology formation process are integrated in the same platform and share the same underlying implementation; it is now possible to develop applications that require all these components. We provide an example of this synergy in the next section.
- Another new feature is the possibility to perform dynamic programming, i.e. dynamic formation-and-execution of the program. We focus on this feature in the rest of this section.

2.1. Dynamic programming with Synergy

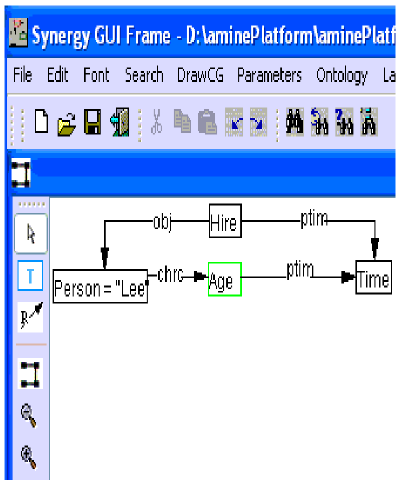
To illustrate what we mean by “dynamic programming”, let us start with the idea of database inference proposed by Sowa [9, p. 312] that combines the user’s query with background information about the database to compute the answer. Background information is represented as type definitions and schemata. Sowa stressed the need for an inference engine to determine what virtual relations to access. By joining schemata and doing type expansions, the inference engine expands the query graph to a working graph (WG) that incorporates additional background information. Actors bound to the schemata determine which database relations to access and which functions and procedures to execute. According to Sowa, his inference engine can support a dynamic way of deriving dataflow graphs [9, p. 312]. In other words, his inference engine can be considered as a basis for *a dynamic programming approach* (recall that dataflow graphs, Petri Nets, executable CG and other similar notations have been used to develop visual programming languages). Indeed, his inference engine is not restricted to database, it can be extended to other domains and be considered as an approach to dynamic programming.

Our task was to adapt, generalize and integrate the inference engine of Sowa to Synergy. The new version of Synergy includes the result of this integration. Figure 1 illustrates the implementation of Sowa's example in Synergy. Background information (procedural knowledge in terms of strategies, methods, procedures, functions, tasks, etc.) is stored in ontology as situations associated to concept types (Figure 1.a). During the interpretation/execution of the working graph (WG) (Figure 1), if a concept needs a value that can not be computed from the actual content of the WG (Figure 1.b), then Synergy looks, in the ontology, for the best situation that can compute the value (i.e. the descriptor) of the concept. The situation is then joined to the WG (Figure 1.c) and Synergy resumes its execution. In this way, the program (i.e. the WG) is dynamically composed during its execution (Figure 1).

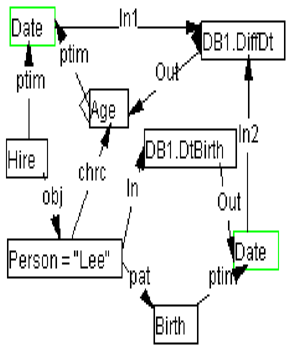
This simple example illustrates the advantage of Amine as an Integrated Development Environment (IDE); it illustrates how various components of Amine (ontology, CG operations, Prolog+CG, Synergy) can be easily used in one application: semantic analysis of the request can be done by a Prolog+CG program. The result (an executable CG), will be provided to Synergy which illustrates the visual execution of the "dynamic program". After the termination of the execution, the final CG will be an input for a text generation program (that can be implemented in Prolog+CG) to provide a text that paraphrases the composed "program" responsible for the result. See Amine Web Site for more detail on dynamic programming with Synergy.



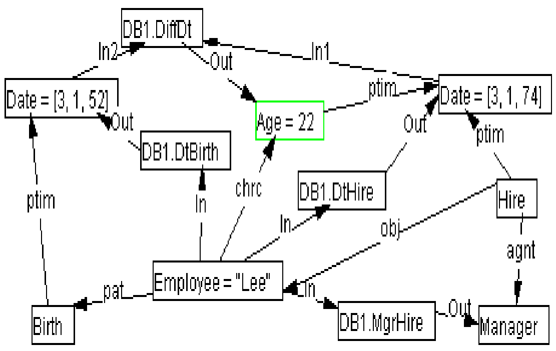
(a) Snapshot of the ontology



(b) The request: initial state of the WG



(c) WG after first maximalJoin



(d) WG after second maximalJoin and termination of execution

Figure 1: Example of Dynamic programming with Synergy (adapted from Sowa [9])

3. Natural Language Processing with Prolog+CG

As stressed in a previous paper [5], several features of Prolog+CG makes it a suitable language for the development of natural language processing (NLP) applications: a) Prolog+CG is an extension of Prolog which is suited for NLP, b) CG, both simple and compound, is provided as a basic data structure, c) Prolog+CG allows CG with variables (variable as concept type, concept designator, concept descriptor, or as relation type), d) several CG matching-based operations are provided (maximalJoin, generalize, subsume, contract, expand, analogy, etc.), e) CG basic operations are available (find a concept or a relation in a CG that verify some constraints, etc.), f) the possibility to construct and update a CG (by adding more concepts and relations).

All these features (and others) are made simpler with the integration of Prolog+CG in Amine platform. Note that with the new version of Prolog+CG, there is also the possibility to use directly the first two layers of Amine. To illustrate the usefulness of all these features for NLP, let us consider briefly their use in three sub-tasks of NLP: semantic analysis, question/answering and phrase/text generation.

Semantic analysis with Prolog+CG

In [5], we illustrated how the above features of Prolog+CG can be exploited to develop a semantic analysis process. As a recall, let us consider the following rule that shows also the use of new features of Prolog+CG. It illustrates : a) the use of variables as concept type, concept designator, concept descriptor and relation type, b) the construction of a concept (E_NP = [N : A1]), c) the construction of a CG (G = [N : A1]-R1->[T1 = V1]), d) the use of the primitive branchOfCG that locates a branch B in the CG G so that B unifies with the pattern given as the second argument of branchOfCG, e) the use of the first two layers of Amine: branch (i.e. a relation with its source and target concepts) and CG are two structures of Amine, these structures with their methods can be used directly in a Prolog+CG program. In our example, we have a call to the method getSourceConcept() that returns the source of the branch/relation and a call to the method specialize() that specializes a CG by the maximal join of another CG.

```
statiivePart([A|P1], P1, G_NP, E_NP, G) :-
   Adj(A, R1, T1, V1), !,
   E_NP = [N : A1],
   G = [N : A1]-R1->[T1 = V1],
   branchOfCG(B, [N : A1]-R1->[T1 = V1], G),
   E_N is B:getSourceConcept(),
   G:specialize(E_N, G_NP, E_NP).
```

Let us consider now the change in the formulation of the lexicon: in previous versions of Prolog+CG, the semantic of the words should be specified in the Prolog+CG program itself. For instance, consider the word “open” with some of its different meanings:

```
lexicon("open",verb,
           [Human]<-agnt-[Open]-obj>[OpenableObject]).
lexicon("open", verb, [Key]<-agnt-[Open]-obj->[Door]).
lexicon("open", verb, [Open_Box]<-agnt-[Open]-obj->[Box]).
lexicon("open", verb, [Shop]<-pat-[Open]-
                                      -obj->[Door],
                                      -ptime->[Time]).
```

With the new version of Prolog+CG, another formulation is now possible: the above different meanings can be considered as background information, stored in the used ontology as situations associated to the type Open. User can access the ontology to get background information (definition, canon, situation, etc.) for a specific type or individual. These changes in the formulation of a lexicon in Prolog+CG lead to the following reformulation:

```
lexicon("open", verb, Open). // one entry for the word "open"

lexicon(_verb, verb, _type, _sem) :-
   lexicon(_verb, verb, _type),
   getSemantic(_type, _sem).
```

Definition of the goal getSemantic/2 is provided below. It searches, from the ontology, the background information for a specific type or individual. Note the call to the method getCanon() that returns the canon of the type (or returns null if the type has no canon) and the call to the method getSituationsDescription() that returns, in a list, all situations descriptions that are associated to the specified type.

```
getSemantic(_Type, _Sem) :-
   _Sem is  _Type:getCanon(),
   dif(_Sem, null).

getSemantic(_Type, _Sem) :-
   _EnumSitDescr  is  _Type:getSituationsDescription(),
   dif(_EnumSitDescr, null),
   _ListSitDescr is
```

```
            "aminePlatform.util.AmineObjects":
                         enumeration2AmineList(_EnumSitDescr),
    member(_Sem, _ListSitDescr).
```

Word disambiguation is performed in the current version of our semantic analysis process by using the backtracking of Prolog+CG: if the maximal join of the word's semantic with the working graph fails, Prolog+CG backtracks and resatisfies the goal getSemantic/2 which returns another meaning (i.e. another conceptual structure) for the current word.

Question/Answering

Semantic analysis of a (short) story would produce a compound CG (see the fragment below). Let us call it CGStory. In our example, CGStory is a fusion of three networks: a) temporal network composed by "after" relations that specify the temporal succession of actions, events, and states, b) causal network composed by "cause" relations, and c) intentional network composed by "motivationOf" and "reason" relations:

```
story(
[Action #act1 =
     [Time : Early]<-time-[WakeUp]-pat->[Man: John]]-after->
[State #stt1 = [Hungry]-pat->[Man: John]]-after->
 ...
[State #stt1]<-reason-[Goal=
            [Action=[Food]<-obj-[Eat]-agnt->[Man:John]]]-
                                     <-reason-[Action #act2],
                                     <-reason-[Action #act5],
                                     <-reason-[Action #act7],
                                     <-reason-[Action #act8]
[Action #act3]<-motivationOf-[Goal =
     [Action = [Man:John]<-dest-[Greet]-agnt->[Woman: Mary]]
                                     ]<-reason-[Action #act4]
[Event #evt1]-
           -cause->[State #stt2 = [ParkingLot]<-pat-[Slick]],
           <-cause-[Event #evt2]
                   ).
```

Semantic analysis process is applied also to questions and for each type of question; there is a specific strategy responsible for the search and the composition of the answer [1]. Here is the formulation in Prolog+CG of the strategy for answering "why" question. It concerns the intentional network: the strategy locates in CGStory the branch/relation with relation type "reason" or "motivationOf" and the branch's source concept should unify with the content of the request. The recursive definition of the goal reason/2 provides the possibility to follow an "intentional path" to get the reason of the reason, etc.

```
answerWhy(A, Y) :-
   story(_story),
   member(R, [reason, motivationOf]),
   branchOfCG(B, [T = G]<-R-[T2 = A], _story),
   reason([T = G], Y).

reason(X, X).
reason([T = G], Y) :-
   story(_story),
   member(R, [reason, motivationOf]),
   branchOfCG(B, [T1 = G1]<-R-[T = G], _story),
   reason([T1 = G1], Y).
```

For instance, to the question "why did john drive to the store ?", the question/answering program returns:

```
?- questionAnswering("why did john drive to the store ?",
                                                _answer).
{_answer = [Goal = [Action = [Eat #0] -
                                    -agnt->[Man :John],
                                    -obj->[Food]
                     ]
           ]};
{_answer = [State = [Hungry]-pat->[Man :John]]};
 no
?-
```

Of course, the above definition of “why-strategy” is simplistic, but the aim of the example is to show how Prolog+CG, in the context of Amine, constitutes a suitable programming environment for CG manipulation and for the development of NLP applications.

Phrase generation

Nogier [8] proposed a phrase generation process that is based on: a) word selection, b) transformation of the input CG to a “syntactic CG” using semantic/syntactic corresponding rules, c) and then linearization of the “syntactic CG” using syntactic and morphological rules. All these rules can be implemented in Prolog+CG. To produce a concise and precise sentence, the generation process has to select the most specific words for the concepts in the input CG [8]. The approach proposed by Nogier can be implemented in Amine as follows: use the dynamic ontology engine of Amine to classify the input CG according to its concepts. The result of the classification, for each concept, is a list of “Conceptual Structures (CS) nodes” in the ontology that are the most close to the input CG. Select from these CS nodes, those that correspond to type definitions. Compute the correlation coefficient proposed by Nogier on the selected definitions to get the “best” words for the current concepts in the input CG. We are developing a phrase generation process that is based on the work of Nogier [8] and that uses the above implementation for the “word selection” procedure.

4. Multi-Agents Systems (MAS) with Amine and Jade: The case of Renaldo

Instead of developing a specific multi-agents layer for Amine, we decided to use available open-source “Java Agent Development Environments” in conjunction with Amine. In her DESA, Kaoutar ElHari explored the use of Amine and Jade[5]. Jade allows the creation of Agents (i.e. it offers a Java class *Agent* and manages the

[5] jade.tilab.com

underlying network processing), the use of different kinds of behaviours and the communication with ACL according to FIPA specification. Currently, we use Jade to handle the lower level of the MAS (i.e. creation and communication between agents) and Amine for the higher level (i.e. cognitive and reactive capabilities of the agents are implemented using Amine).

Currently, the MAS layer of Amine contains one plug-in (Amine/Jade) implemented as a package: *amineJade*. Other plugs-in (i.e. other packages) could be added as other combinations of Amine and “Java Agent Development Environments” are considered (for instance Amine and Beegent[6]). The package “amineJade” offers basically two classes:

- The class *PPCGAgent* that extends the class Agent (provided by Jade) with Prolog+CG interpreter (as its main attribute) and with other attributes and methods (like *send*, *sendAndWait* and *satisfyGoal*).
- The class *JadeMAS* that offers the possibility, via the method *createMAS*, to create and initiate a multi-agents system.

Let us consider briefly the case of *Renaldo*; a MAS that concerns the simulation of a child story. The setting of the story is a forest; it corresponds to the environment of Renaldo. The characters of the story (the bear John, the bird Arthur, the bee Betty, etc.) are the agents of Renaldo. Each type of agents (bear, bird, bee, etc.) has a set of attributes, knowledge, goals, plans and actions that are specified as a Prolog+CG program. A specific agent can have, in addition, specific attributes, knowledge, goals, plans and actions, specified also as a Prolog+CG program.

The MAS Renaldo is implemented as a Prolog+CG program/file: “Renaldo.pcg”. The execution of the MAS Renaldo is initiated by calling the goal “renaldo” which is defined as follows:

```
renaldo :-
   "aminePlatform.mas.amineJade.JadeMAS":createAgents(
                         [John, Arthur, Betty, Environment]),
   John:satisfyGoal(
       goal([Bear: John]<-pat-[SatisfyNeed]-obj->
                  [Hungry]-Intensity-> [Intensity = High])).
```

The argument of the method createAgents() is a list of agents identifiers. From the identifier of each agent (i.e. John), the method gets the associated Prolog+CG program/file that specifies the agent (i.e. “John.pcg”). It then locates the header fact of the agent to get the ontology and names of other Prolog+CG programs associated to the agent. For instance, the header of the agent John (from the program “John.pcg”) is:

```
   header("RenaldoOntology.xml", ["Bear.pcg"]).
```

The method createAgents() will then create an instance of PPCGAgent class for each agent and initiates the associated Prolog+CG interpreter with the specified Prolog+CG files and ontology file. For instance, createAgents() will create an instance of PPCGAgent class for John and will initiate its Prolog+CG interpreter with the files “John.pcg” and “Bear.pcg”, and with the ontology “RenaldoOntology.xml”.

[6] www2.toshiba.co.jp/beegent/index.htm

Note: the environment is implemented as an agent that manages the access, by the agents, to shared objects (resources like foods, water, river, etc.) and it is responsible also for the treatment of events.

After the creation and initiation of the agents (due to the execution of the method createAgents()), “renaldo” assigns to John the goal “satisfy hungry with intensity high”. The method satisfyGoal() of PPCGAgent calls the Prolog+CG interpreter of the agent (recall that each agent has its own Prolog+CG interpreter) to resolve the specified goal.

Renaldo in particular and Amine’s MAS layer in general will be described in more detail in a forthcoming paper.

5. Related works

Philip Martin[7] provides a detailed comparison of several available CG tools[8] (Amine, CharGer, CGWorld, CoGITaNT, Corese, CPE, Notio, WebKB). CGWorld is a Web based workbench for joint distributed development of a large KB of CG, which resides on a central server. CGWorld is no more developed. Corese is a semantic web search engine based on CG. WebKB is a KB annotation tool and a large-scale KB server. CoGITaNT is an IDE for CG applications. CharGer is a CG editor with the possibility to execute primitive actors and to perform matching operation. Notio is not a tool but a Java API specification for CG and CG operations. It is no more developed. It is re-used however by CharGer and Corese. CPE has been developed as a single standalone application. Currently, CPE is being upgraded to a set of component modules (to render CPE an IDE for CG applications). Its author announces that CGIF and basic CG operations (projection and maximal join) are coming soon. The new upgraded version of CPE is underway and it is not yet available. CGWorld, Notio and CPE will not be considered in our comparison of available (and active) CG tools.

In his comparison, Philip focuses mainly on the “ontology-server dimension” which is specific to his tool (WebKB); he did not consider other dimensions, i.e. other classes of CG tools. Indeed, CG tools can be classified under at least 8 categories of tools: CG editors, executable CG tools, algebraic tools (tools that provides CG operations), KB/ontology tools, ontology server tools, CG-based programming languages, IDE tools for CG applications and, agents/MAS tools.

The category “IDE for CG applications” means a set of APIs and hopefully of GUIs that allow user to construct and manipulate CGs and to develop various CG applications. Only Amine and CoGITaNT belong to this category. The category “CG-based programming language” concerns any CG tools that provide a programming language with CG and related operations as basic construct. Only Amine belongs to this category, with its two programming languages: Prolog+CG and Synergy. The category “Agents/MAS Architecture” concerns CG tools that allow the construction and execution of intelligent agents (with cognitive and reactive capabilities) and

[7] en.wikipedia.org/wiki/CG_tools

[8] These tools are listed also in www.conceptualgraphs.org/

multi-agents systems (MAS). As illustrated in this paper, Amine, in conjunction with a Java Agent Development Environment, can be classified under this category.

Symbols used in the table:

"++": the tool offers different features concerning the associated category. For instance, Amine provides multi-lingua and multi-notations CG editors. The same for executable CG: it offers not only the equivalent of actors, as CharGer does, but a programming language based on executable CG. This paper illustrates in addition a new feature of Synergy: dynamic programming. The same for "KB/Ontology" category: Amine provides a rich ontology API, ontology editors and various basic ontology processes. And the same for "Programming" category: Amine provides two CG based programming languages (i.e. Prolog+CG and Synergy).

"+": the tool can be classified under the associated category.

"-": the tool can not be classified under the associated category.

"/": the tool is not intended to belong to the specified category but it uses some aspects of the category. For instance, "web ontology tools" like Corese and WebKB are not intended to be used as "algebraic tools" even if they use some CG operations (like projection and generalization).

	Amine	CharGer	CoGITaNT	Corese	WebKB
CG Editor(s)	++	++	++	-	-
Exec. CG	++	+	-	-	-
Algebraic	++	+	+	/	/
KB/Ontology	++	-	?	+	+
Ont. Server	-	-	-	-	++
IDE	++	/	+	-	-
Programming	++	-	-	-	-
Multi-Agent	+	-	-	-	-

Figure 2: Comparison of available CG tools

6. Current and future work

Current and future works concern all layers of Amine as well as the development of applications in various domains:

a) Development of the ontology layer: development of interfaces with ontologies that use RDF/OWL, development of Web services so that Amine ontologies can be used from the Web, development of an ontology server, enhance the current ontology drawing module, enhance the basic ontology processes, etc.

b) Development of the algebraic layer: enhance the implementation of CG operations, consider other implementations, enhance the CG drawing module, etc.

c) Development of the programming layer: enhance the debugger of Prolog+CG as well as its interpreter and its GUI, complete the implementation of all the features of Synergy, etc.
d) Development of inference and learning strategies, that will be considered as memory-based strategies to provide an *operational memory-based and multi-strategy learning programming paradigm,*
e) Development of several applications in various areas: reasoning, expert systems, ontology-based applications, natural language processing, problem solving and planning, case-based systems, multi-strategy learning systems, multi-agents systems, intelligent tutoring systems, etc.

Conclusion

Amine platform can be used to develop different types of intelligent systems and multi-agents systems, thanks to its architecture; a hierarchy of four layers (ontology, algebraic, programming and agents layers) and to the “openness” of Amine to Java. This paper illustrates the use of Amine in the development of dynamic programming applications, natural language processing, and in multi-agents systems applications. The companion paper illustrates the use of Amine in ontology-based applications.

We hope that Amine will federate works and efforts in CG community (and elsewhere) to develop a robust and mature platform for the development of intelligent systems (including semantic web) and multi-agents systems.

References

[1] Graesser A. C., S. E. Gordon, L. E. Brainerd, QUEST: A Model of Question Answering, in Computers Math. Applic. Vol 23, N° 6-9, pp. 733-745, 1992

[2] Kabbaj A., K. Bouzouba, K. ElHachimi and N. Ourdani, Ontologies in Amine platform: Structures and Processes in Amine, submitted to the 14th ICCS 2006.

[3] Kabbaj A., Amine Architecture, submitted to the Conceptual Structures Tool Interoperability Workshop, hold in conjunction with the 14th ICCS 2006.

[4] Kabbaj A. and M. Janta, From PROLOG++ to PROLOG+CG: an object-oriented logic programming, in Proc. Of the 8th ICCS'00, Darmstadt, Allemagne, Août, 2000.

[5] Kabbaj A. and al., Uses, Improvements and Extensions of Prolog+CG: Case studies, in Proc. Of the 9th ICCS’01, San Francisco, August, 2001.

[6] Kabbaj A., Synergy: a conceptual graph activation-based language, in Proc. Of the 7th ICCS'99, 1999.

[7] Kabbaj A., Synergy as an Hybrid Object-Oriented Conceptual Graph Language, in Proc. Of the 7th ICCS’99, Springer-Verlag, 1999.

[8] Nogier J-F., *Génération automatique de langage et graphes conceptuels,* Hermès, Paris, 1991.

[9] Sowa J. F., *Conceptual Structures: Information Processing in Man and Machine,* 1984, Addison-Wesley.