Design Decisions for the Incremental A*dage* Framework

*Jean-Louis Giavitto Guy Rosuel Agnès Devarenne Anne Mauboussin**

Laboratoires de Marcoussis - CGE Corporate Research Center
Route de Nozay
91460 Marcoussis, FRANCE

Abstract

A*dage* is an incremental software development support environment**. A*dage* architecture is underlined by the concept of wide-spectrum services around a common data model, the GDL. This model extends the entity-relationships model and provides useful concepts to support genericity and incrementality like inheritance and self-reflexion. Services respond both to end-user and administrator requirements. The last point is achieved through an active help for tool integration and tool building and so for making the environment evolve to cover the needs.

1 Introduction

A*dage* (**A**d**a**ptable **g**raph-based **e**nvironment) is an incremental software development support environment (SSE). The aim of a SSE is twofold:

- to integrate the different tools and products used and produced through the different phases of the software life cycle, e.g. to provide facilities to represent the various entities involved in the software development process;
- to enable the management and the evolution of these entities, by introducing a comprehensive support for the user's actions, monitoring the tools and maintaining the constraints.

The kind of data an environment can manage, is strongly related to the phases of the life cycle it addresses. **Generic** environments, as opposed to ad-hoc environments, have the capacity to define the data handled by the system. The ability to cover different activities depends on the expressive power of the data model. Nevertheless, an adequate data model is not sufficient to define a generic SSE. A framework must also be provided, where tools are integrated and where users and tools can cooperate. Moreover such an integration must be **incremental** following the natural evolution of the target environment.

Tool to tool cooperation is a difficult problem: Each tool has its own point of view on the data it has to manage. And even if tools share a common point of view on entities, each one has its own internal representation of them.
User to tool cooperation, and more generally users to environment cooperation, is usually left back in user-interface area. But more than graphical gadgets, the user-interface of an environment has to provide useful and suitable concepts to browse data, trigger tools in a cohesive and uniform way, manually create and update entities...

Therefore, in a multi-users, multi-tools environment, data accesses and data updates must be monitored to avoid inconsistency. Nevertheless development should proceed in parallel as far as possible.

The purpose of this paper is to expose the motivations of the A*dage* project and the consequent design decisions. We present in the following sections the goals of the project, the architecture of the A*dage* environment, the underlying data model, the status of the user-interface and of the tools and finally the concurrence problems which arise in a multi-users multi-tasks framework. Orthogonal to these investigations, *incrementality* and *applicability* (choice of the right technology), which influenced the design, appear all along the text.

2 An incremental environment

Genericity is the ability to generate an environment from a description. The benefits of this approach has been largely demonstrated, see for example Alma [23], Eclipse [13], Concerto [28], GIPE [9]... The nature of the final environment depends on the kind of data that can be described. For example, in GIPE, several description languages exist to describe grammars and manipulate syntax trees (LDF). It enables the generation of language centered environments. Here, the genericity lies in the independence of a language, but not of life cycle phases. What we wanted to produce was a software factory dedicated to *programming in the large*, independent of languages, methods and tools. From this point of view **A*dage*** [19] belongs to the family of environments such as GraphTalk [21] and The Virtual Software Factory [30]. **A*dage*** relies on a data model able to describe the data occurring within and exchanged between the various phases [11].

---

* Consultant from LIF, Université Pierre et Marie Curie, 4 Place Jussieu 75252 Paris Cedex 05.

** This project has been partially supported by the Esprit project METEOR P432 and is supported by ALCATEL funding.

0270-5257/90/0000/0086/$01.00 © 1990 IEEE

Recommended by: Dick Fairley

The data model has to handle various kinds of software entities and relationships [20]: This naturally leads to the use of a kind of entity-relationship model [8]. It induces some ideological consequences, e.g. the development process is data-oriented and not activity-oriented. Moreover, the implementation of the model, the *information system*, constraints the data that can be reasonably managed.

Related to the various activities, several formalisms can cohabit and must be supported together. Genericity solves this problem of profusion of needs. Besides, data organization and tools are evolving in time and the environment must be able to follow them. *Incrementality* improves the concept of genericity: At any time, the environment can be extended to handle evolution. Incrementality is a key to secure the future.

Producing an incremental environment w.r.t. the previous motivations implies multiple design decisions: choice of the data model, choice of an architecture, choice of task management, etc.

3 Wide spectrum services

A data model theoretically handles *all* the entities appearing in a software development process. However, the underlying implementation sets a barrier: the granularity of the data that are pertinent to manage. The problem is not to draw back these representation limits but to integrate the tools that fill the holes in the description of the entities. For example, it is not reasonable for a programming in the large environment to represent the syntax tree of a program; the environment delegates this work to the compiler. What we do need are ways to build, connect and integrate tools. The last point means, in particular, to be able to handle the interaction between entities *owned* by the environment and entities known by the tools.

So we prefer to see an environment as a set of services rather than a set of fixed tools. The concept of service plays the same role for the architecture as the genericity for the data: Services can be extended, reused and specialized to fit evolution and needs. A service is defined through three points of view, the user of the service, its nature and its domain.

3.1 A*dage* services

The A*dage* workbench is not only intended for the end-user of the environment, but addresses tools, environment builders, administrators, expert and naive end-users. Naturally, competences may overlap and for example, an expert user often increases the common set of services provided by the administrator with its own tools. Services provide help to build tools in the following domains: data storage, data management and data visualization.

A*dage* services go from basic libraries (persistency and graphical display) up to ready-for-use tools. A tool can be integrated at different levels according to the services it requires. A priori, the higher the used services are, the easier the integration is. The coupling between a tool and the framework is illustrated in figure 1.

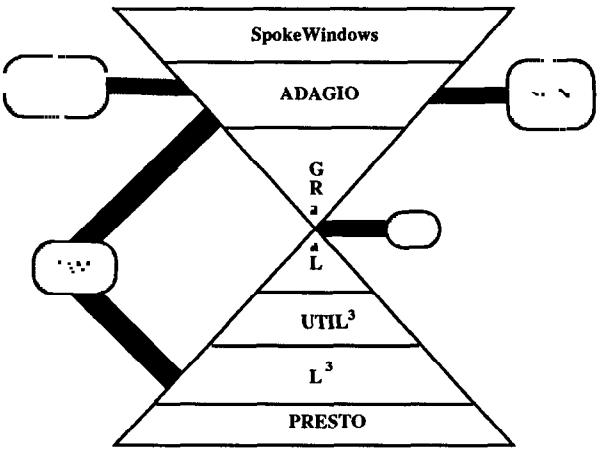

Fig. 1: The A*dage* framework as an ecosystem for the tools. ADAGIO is used to display graphs for speech recognition in the SPIN Esprit project and to display version graphs in the PEGASE environment [6]

3.2 Classes of services

Higher A*dage* services use lower services. The services have different interfaces; we describe them very briefly, some of them are detailed in the rest of this paper:

- **Generic libraries:**
  PRESTO (see §5.3) is a generic library introducing persistency in the C++ language. A second one is a portable window manager, *SpokeWindows* [22]: It ensures a common graphical shelf based either on X or on Sunview displays [16] [29].

- **Kernel libraries:**
  On top of the generic libraries, two kernel libraries are built, L^3 (see §5.4) and *ADAGIO*. L^3 implements A*dage* entities. ADAGIO extends SpokeWindows to display graphs.

- **"Compilers":**
  The *GDL incremental compiler* (see §4) translates a textual description of A*dage* entities into data owned by the environment. *UTIL*^{3} is used to maintain constraints among the data. It compiles a description of the changes that have to be monitored by the information system, to customize the generic tools.

- **Generic tools:**
  Graal (see §6) is the standard A*dage* shell and it is used to both manage data and provide a graphical user-interface. *JaM* (see §8) is the tool used to manage workspaces.

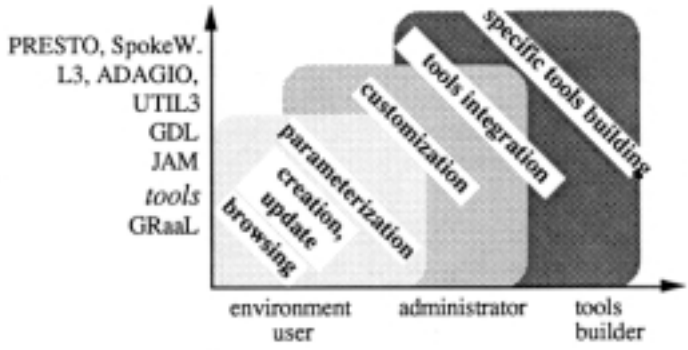

Fig. 2: Who does what and how.

- **Instantiations**:
  Some A*dage* instantiations are interesting enough to be embedded as additional services. For example, the documentation of A*dage* is represented through A*dage* entities managed in the A*dage* framework.

Figure 2 is an attempt to map users, services and activities. We qualify as wide the A*dage* services because they address several users and concern several activities.

4 A specific data model

4.1 An extension of the Entity-Relationship model

The A*dage* data model had to fit well the software engineering needs, being able to represent the main characteristics of the data managed by a programming in the large environment. Multiple objects are involved, often numbering in the thousands. Elementary objects, e.g. operating system resources, have to be gathered into consistent entities, because they are different facets of pertinent *abstractions* for software developers. Unfortunately, such entities are meaningful only in the developer's mind and exist virtually only as long as the developer maintains coherences between the basic objects.

Each tool sees entities from its own viewpoint, i.e. it uses only a subset of its facets. For example, the abstraction "*module*" consists of a documentation, a source file, an object file..., that must be managed in an integrated way because they all represent the same logical entity with a well defined meaning for the end-user. The documentation is the point of view of the *module* that concerns the documentation editor, while the source and object files concern the compiler, as illustrated in figure 3.

To be useful, abstractions must be able to be specialized: a *Cmodule* is also a *module*; and entities must be aggregated: a specification is also the set of the used sub-specifications. Aggregation, composition, hierarchy, refinement are the way to structure the world of the entities. All these characteristics can be summed up by saying that the concerned objects are complex and structured entities, with many arbitrary relationships between them. The goal of the data model in a SSE is to represent such entities and relationships; it is the purpose of the A*dage* framework to support the life cycle of theses objects.

Standard uses of the Entity/Relationship model suffer of too many lacks to represent such objects [26]:

- Data structuring is too flat, composition and specialization do not exist;
- Complex attributes are not allowed;
- Viewpoints are rather sub-schemas than real structuring;
- Hierarchical decomposition is uneasy, refinement needs an explicit relationship between the refined object and its refining components.

That led us to extend the standard E/R model into a model based upon structured typed graphs. Graphs are a universal concept to model objects and relationships. The A*dage* data model increases the expressive power of the E/R classical approaches with powerful means of data structuring. Our model is embodied in a language, the Graph Description Language (**GDL**). Other projects, as Damokles [10] [15], have already adopted this kind of models but without providing *specialization* and *self-reflexion* (see below).

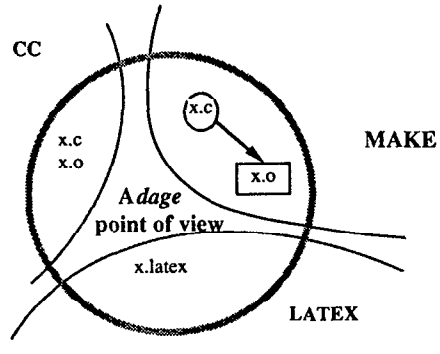

Fig. 3: A module in the A*dage* point of view versus the tools point of view.

4.2 GDL nodes

The basic GDL entities are called **nodes**. A node exists by itself and can be manipulated in a global way. It is described on one side by its specific properties and on the other side by its interactions with the other nodes.

The node in itself: its attributes

The **attributes** of a node describe its intrinsic properties. For example, every C module has an associated text file. This text file is an attribute of the node implementing the module concept. Furthermore, an attribute is a way to connect a node with existing tools and resources. For instance, declaring the attributes "source" and "object" of type "file" in a module makes it accessible to an existing compiler.

Attributes are named functional relationships between a node and typed values. These attribute values belong to the types *boolean*, *integer*, *file*, *date*, *bounded string*, and *node*. *Node* means that the type of an attribute can be that of a node (see further for node typing): This gives the possibility for a node to refer to another node through an attribute. The future version of the GDL will include type constructors as sets, multisets and lists.

The node in the world: its graph

The **graph** part of a node represents its interactions with other nodes. A graph is composed of **nodes** (its vertices) and **arrows** (its edges). An arrow can link a source to a target node and so establishes a binary relationship between two entities.

Vertices of a graph are nodes: this is a structuration mechanism for A*dage* entities. This kind of structuration is more general that just a strict hierarchy. It can be used to represent functional decomposition or more complex aggregation relationships. A same node may appear in several graphs as it may be seen according to different viewpoints.

Arrows are very thin entities. They have neither name nor attributes (but they have a type). They are just a light link between two nodes in the context of another node. Heavy or *n*-ary relationships will be described through the use of additional nodes. The semantics of an arrow is very broad: it can represent a constraint, an aggregation, a "part-of" link or any kind of relationships (function calls, include dependencies, etc.).

A powerful structuring

In the A*dage* model, relationships, as well as attributes, are encapsulated inside the nodes, in contrast with PCTE or usual extensions of the E/R model where they are objects of the same level as the entities. That offers rich structuration means. Small specialized graphs and sub-graphs show immediatly numerous different points of view upon the database in a much more efficient way than in big flat graphs where filtering is necessary.

4.3 Typing

Each node is typed: The type of a node describes its structure, i.e. it specifies the attributes types and which types of nodes and relations can belong to its graph. Therefore, the type of a node is also represented as a node. Thanks to this property (**self-reflexivity**), types and instances are manipulated in a uniform way. The creation of an entity is done through the instantiation of its type. The type allowing the creation of other GDL types is called T; T describes its own type.

Inheritance mechanisms are useful to make very fine-grained descriptions. The GDL sub-typing (or **derivation**) must be seen as a kind of specialization (similar to single inheritance in the object paradigm). Thus, if T2 is derived from T1, all instances of T2 have the properties described by T1. A type, derived from an existing one, defines additional attributes and specializes the base graph. An arrow is derived by specializing its source and its target. All the GDL types are derived, directly or transitively, from the weakest type called U. U is also derived from U. A graph typed U has no attributes and possesses any kind of graph.

5 Implementation of the data model

5.1 Extendability

In a flexible framework like A*dage*, it must be possible to define new types of data to take into account any evolution of the outside world: new organization, adoption of new tools... This can be achieved in A*dage* through the derivation and the self-reflexion properties: Adding a new type is done by extending an existing one (derivation) and this is possible because types are accessible in the system as data (self-reflexion).

Therefore we built an incremental GDL compiler. The GDL compiler is responsible for translating a textual description of nodes into data owned by the system. Adding new types (or new instances of existing types) is just compiling a GDL program unlike many database management systems (DBMS), where extending a relationship forces to compile the whole schema again.

5.2 Granularity issues

What should be modeled, what kinds of objects should appear in a GDL modelization ? This problem is related to granularity of the description: the smallest object that can be managed. In a fine-grained description, the objects taken into account can be very small and the modelization very accurate (but "accurate" is not "pertinent" nor "useful"). One of the main motivations of our design is the ability to implement entities lighter than a file but heavier than a tuple in a relational database. They will be for example SADT boxes, PERT diagrams, SDL schemas, call graphs or hypertext structures...

```
module : T :: U {
   attributes :
      doc : documentation;
      owner : string32;
   graph :
      depends_on ::R :
           module->module;
};


Cmodule : T :: module {
   attributes :
      source : file;
      object : file;
   graph :
     includes :: depends_on :
           Cmodule->Cmodule;
     calls :: depends_on :
           Cmodule->Cmodule;
};
```

```
documentation : T :: U {
   attributes :
        text : file;
};


hash_code : Cmodule {
   attributes :

      doc : hash_doc;
      owner : "john";
      source : #hash_code.c;
      object : #hash_code.o;
   graph :
     includes :
       hash_code->string,
       hash_code->stdio;
     calls :
       hash_code->string;
};
```

Fig. 4: A GDL declaration.

In this toy example, we define a module as an entity with an associated documentation and an owner. The graph we associate arbitrarily with a module represents its direct dependancies. So, an instance of a module appears in its graph as the source of all the arrows, the targets of these arrows are the modules used to built the previous one.

Three types are defined: *module, documentation, Cmodule* ; *hash_code* is an instance of *Cmodule*. *module* and *documentation* are instances of *T* and inherit from *U*. *Cmodule* is an instance of *T* and is a specialization of *module*. The arrows of a module are of one type, *depends_on*, a specialization of the relation *R* the universal relation inherited from *U*. *depends_on* is specialized in *includes* and *calls* for *Cmodule*. The value of the attribute *owner* of *hash_code* is *john*. The vertices of *hash_code* are *hash_code* itself, *string* and *stdio* (we assume that *string* and *stdio* types are sub-types of *Cmodule*).

5.3 Persistency

GDL entities must be implemented in a persistent fashion. A solution could have been to put an interface on top of a classical (relational) DBMS [24]. But the relational paradigm is not well suited for storing persistent GDL graphs, because of the following drawbacks:

- A graph representation must be split in several tuples, loosing its perception as a whole and making its manipulation difficult and inefficient ;
- A tuple exists through its values while a node needs to have an identity, i.e. a proper existence independently of its graph and the value of its attributes at a given time;
- A usual relational database is well suited to manage a lot of tuples of few types; on the opposite, the *Adage* environment is expected to manage many types;
- Navigation in relational databases is made by iterations upon tuples while browsing activities, very numerous in SSEs [17], needs navigation from object to object following the relationships that link them.

As a matter of fact, traditional DBMSs simply fall far short of providing a viable base on which computer aided-design, software environment or any intensive data application can be built [14][10].

An object oriented DBMS would have been more suitable. But the relational aspects of graphs would have been costly to manage. Another problem is that the kernel libraries that implement GDL concepts are part of the *Adage* services. Thus it is important to reduce the gap between the application programs and the data management functionalities. This problem is well known under the term *impedance mismatch* [25].

For all these reasons, the GDL implementation is supported by adding persistency [4] to the system language (the language giving access to the kernel). This strategy differs from the usual one as in PCTE [27] or CAIS: the use of an external data manager through a functional interface. If the functional interface makes the kernel accesses easier from various programming languages, the advantage of our approach is to remove all the impedance mismatch problems by embedding the data management in the system language.

The system language for *Adage* is C++ [31]. **PRESTO** (**PeR**sistent **E**lementary **S**wappable **T**iny **O**bjects) is the package that adds persistency to C++ [18]. In PRESTO, the persistent objects are those derived from a specific class called "resource". The class descriptions are independent from the persistency concept, providing transparency to C++

programmers. Moreover, thanks to the encapsulation features offered by C++ (*private, protected* or *public* visibility closes, *const* qualifier...), the security ensured by the kernel interface is equivalent to the security that can be enforced through a functional one.

The concurrent access problems are not taken into account by PRESTO. They will be handled by the *workspaces* management described in the section *working in team.*

5.4 L^3: the Low Level Library

The Low Level Library (L^3) is the implementation of the GDL entities upon PRESTO objects: storage of the attributes, of the graphs... It assures some constraints satisfaction such as referential integrity: When a vertice is suppressed in a graph, all the pending relationships are also deleted. L^3 also maintains the inverse of some structural relations to implement efficiently some requests: For example, maintaining the inverse of "belong" speeds up the computation of all the graphs containing a given node.

6 GRaaL

The set of data managed through A*dage* can be manipulated through the language named **GRaaL** (Graph Request Language). The design of GRaaL is driven by three goals:

- to offer standard arithmetic and graph arithmetic,
- to build, encapsulate and trigger tools,
- to provide a graphical representation for A*dage* entities.

The language is supported by an interpreter: **graal**. This interpreter acts as a *glue* between the tools, provides a graphical user-interface and implements the traditional concept of request language in database area.

GRaaL can be seen as a language managing persistent data, where the data definition part is handled by the GDL. The GDL part may be explicitly called in a graal program, to create a node for example. GRaaL is a language based on the notion of function and its top-level consists in reading, evaluating and printing expressions. As usual with interpreted languages, type checking is delayed through the running of the interpreter.

6.1 Arithmetics in GRaaL

GRaaL expressions are used to compute new graphs, to browse among data, to retrieve information and so on. Expressions involving side-effects (attributes affectation, insertion or deletion of a vertice or an arrow) are used to update GDL entities. That is why GRaaL includes arithmetics on simple types (integer, boolean, bounded string, file, date) and on nodes.

Expressions are built with system defined- and user's defined functions. Functions are first citizen values in GRaaL and, for example, they can be arguments of other functions. The capacity for a user to define its own functions (and to call them recursively) enables the computation of transitive closures and the use of powerful graph algorithms like the recursive descent. Such requests are useful to compute for example the set of entities depending on a node through a given relation. They are not provided in usual relational algebra but exist in some extended DBMS [7] [1].

The graph arithmetic can be divided in two parts. On the one hand, the simple arithmetic allows the user to know if a node belongs to a graph, the number of nodes or arrows in a graph, and so on. On the other hand, the complex arithmetic computes new graphs through union, difference, intersection, copy... of graphs. The result is an *ephemeral* graph that exists only during the current graal session. The type of a computed graph is `U` the less constrained node type.

6.2 Streams

Browsing through the organization is done by means of **iterators**. Iterators are a kind of **stream** [3]. Built-in iterators list nodes satisfying a basic criterion: a given name, a given type, the membership of a given graph, source or target of a given vertice in a given graph, etc. User's iterators can be built by filtering and/or by applying a function to each element of an iterator.

Partial application of user's functions (building a new function by instantiating some arguments of another function) is very powerful to create new iterators (see figure 5).

Another kind of stream is used to monitor graal input/output. This includes reading and writing files and is also the standard mechanism to trigger tools and to communicate with them.

```
sum-graph = lambda (sum, inc)
              { sum = +(sum, clone(inc)) };

flat = lambda (x, result)
         { for-node(x) | sum-graph(result),
              result };

flat-module = lambda (x)
              { if (x :< @module,
                    flat (x, new-u()),
                    ?("arg not a module\n")) };
```

Fig. 5: A GRaaL program

With the data defined in fig. 4, the function *flat-module* creates the graph result of the union of the vertices of the argument. This can be used to obtain the dependence graph of a module. *flat-module* verifies first that its argument is of a sub-type of *module*. The function *sum-graph* takes two arguments and substitutes the union of the two for the first. The partial application of *sum-graph*, "sum-graph (result)" in *flat*, produces a function that adds its argument to the graph *result*. This function gets its successive arguments from the iterator *for-node*.*for-node* lists the vertices of a graph.The function *clone* is used to copy a graph. *new-u* creates an empty ephemeral graph. The question mark operator prints a value.

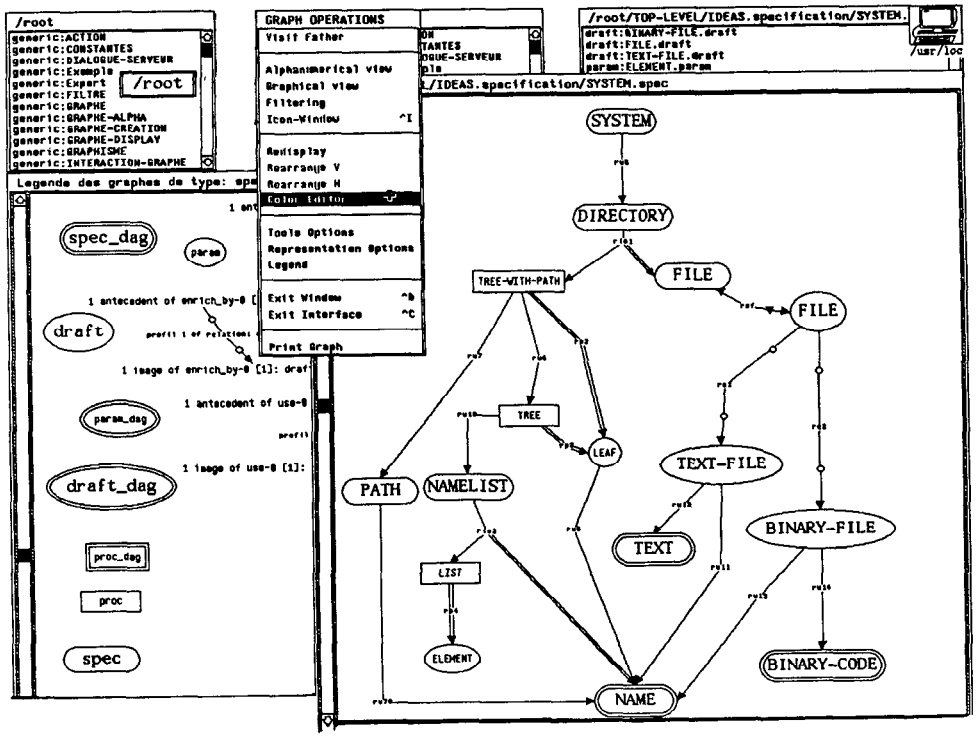

Fig. 6: Hard copy of the window owned by the user interface

6.3 User-interface

The last kind of stream supplies graphical interactions. By default, the graal top-level is connected to a *listener*. The listener provides a textual input/output of graal values. At any time, it can be changed and connected to a window-stream to display graphically the results of the expressions. Such a display is the support for pop-up menus, mouse driven selections, etc.

Incrementality is introduced into the user interface with two features: **styles** and **user's menus**. The graphical appearance of a node, its style, depends on its type. Thus, it can be represented as the list of its vertices, as the value of its attributes, as the graphical drawing of its graph, as the viewing of one of its file attributes and so on. The style of a node can be changed on-the-fly.

System menus are composed of syntactic operations, as nodes layout, and generic operations, as cut and copy. User's menus propose a set of user's selected GRaaL functions. They are used as short-cuts to call these functions on the current selection. Menus are linked with styles and thus, only the correct operations (w.r.t. the type of the selection) are accessible to the user.

7 Tools integration and tools building

The primitive activities are usually performed by invoking tools. Tools fall in several categories: configuration management, document preparation, language support, project management and so on. Openness and flexibility are achieved by enabling the building and the integration of new tools, to face new needs, without altering the architecture and the pre-existing tools. Our intention is to provide a high degree of integration of tools in the *Adage* environment.

Each tool manages its objects according to an internal object model and stores them according to its own representation. Generally, a tool can be considered as an entity providing products for- and using products provided by other tools [32]. To integrate tools, the formalism of the database must be adapted to the data formalism required by tools: This is identified as a *communication* mechanism. It differs according to whether tools are pre-existent or built.

7.1 Pre-existing tool integration

The integration of the pre-existing tools is done via a translation into a *private representation.* So, the translation is the way to embed existing tools and is carried out through GRaaL. A naive illustration can be the generation of the "makefile" corresponding to some relationships between nodes in order to use the unix "make" tool. The general mechanism to trigger a tool consists in getting the data from A*dage* and converting them into the tool data representation. Then the tool runs and the results are possibly collected and converted into the A*dage* framework. Moreover, to ensure entities consistency, the tool effects have to be propagated.

The user-interface and GRaaL are services supporting tools integration (common user-interface, data collecting and translation). **UTIL^3** (**U**ser-action **T**riggering **I**n **L^3**) implements daemons on top of L^3 for constraints propagation and for actions triggering in response to data updates.

7.2 New tools building

Building a new tool can elude the problem of data representation: A new tool should adopt the A*dage* representation of the data; integration is achieved by *sharing of representation.* This is done using L^3 and UTIL^3. A graphical library, **ADAGIO** (A*dage* **I**nput **O**uput), extending the SpokeWindows window manager is also provided to represent graphs easily.

Tools can be built in GRaaL for prototyping purposes or if efficiency is not the critical point. The benefits of using GRaaL are multiple: powerful control structures for data browsing and graph manipulations, interactivity of an interpreted language, integration with graphical interaction and so on.

7.3 Using tools from the environment

The interface between the different tools must be standardized in order to use them easily: A good integration reduces the effort of adaptation by the user and rationalizes the cost of the development of a tool producer.

GRaaL is the standard A*dage* command language. It plays the role of a shell for a tool, verifying the conditions of application and the correctness of arguments, before triggering it.

8 Working in team

Considering the information sub-system as a monolithic server makes the concurrent access management easier but is a bottleneck for the data management. This is moreover increased by the fact that most of the software development transactions are long-lived. It is why the concept of a centralized server, although suitable for business applications, is not applicable here. A solution could be to split and distribute the server among the entities: each entity being responsible for processing the requests it is concerned with, as in the actor paradigm. But this is unmanageable because of the too many servers. Moreover, which entity should be in charge of answering a request concerning several entities?

8.1 Workspace

A better response is brought by the concept of **workspace**; we call workspace a set of entities involved in a task. Tasks are related to the unavoidable division of the work in big projects. Following NSE trends [2], the concurrence control scheme consists in making the task proceed locally on a copy of the entities needed for its achievement. The needed entities are copied only when required in the task private workspace. When the results are available, they must be integrated in the repository of the father task. This paradigm favours contractual working opposed to data sharing. Such a scheme is very flexible. For example, A*dage* workspaces are able to emulate the ISTAR contract database concepts [12].

Unfortunately, a task is realized using multiple tools and the problem of monitoring concurrent activities is not solved and even worse, is duplicated. Indeed; tools have to be monitored on each workspace and workspaces have to be synchronized.

8.2 Intra workspace management

Monitoring tools on each workspace is solved by enforcing a two points policy. On the one hand, workspaces cannot be shared between user-interfaces. On the other hand, a user-interface can share its current workspace with the tools it triggers but they are serialized. This strategy is not ensured by tools themselves but by user-interfaces. However, stand alone tools can use a standard service to subscribe the same behavior. Note that graal is carefully designed to avoid interferences with tool working.

Tools that must run concurrently have to work on separate workspaces. The multiplicity of workspaces is not a problem because they are relatively light objects (five files in the current implementation).

This policy is very strict but avoid the multiplication of lock mechanisms. As a matter of fact, workspaces are the counterpart of the lock mechanisms (as, for example, implemented in PCTE), with the advantages of not pre-empting ressources but with the inconvenient of the synchronization problems.

8.3 Inter workspaces management

Workspaces are organized in a hierarchical tree. A father workspace is the integration area for all its child workspaces. If an entity is not present, it is searched in the ancestors and copied. Later, workspaces must be joined up and this is done by an integrator service. So a workspace is characterized by three parameters:

- The father is used to search an entity, and that recursively.
- A copy function interns an entity in the current workspace. Because the entities can refer complex attributes, copy functions can differ by the level of the copy. As an example, consider the status of a file attribute: It can be shared between versions of the same entity or private to each version.
- A join function merges the entities to its father workspace.

Different schemes for object management exist, from the safest to the most lax. The safest scheme is based on a derivation/lock model [5] and ensures, in addition to the concurrency control, a versioning mechanism. Each merge of workspaces creates a new version of the involved entities, keeping so a trace of the development. For that, a full copy function is employed. The merge function should be optimized not to duplicate unchanged objects. The lax scheme implements the fact that the last merged workspace "wins", like usual text editors running on the same file. An intermediate issue consists in a full copy and a user-directed merge through a tool similar to the unix "diff". This tool presents interactively to the user the pair of conflicting entities and it is his responsibility to merge them cleverly.

We can note that graal integrates the concept of workspace at the user-level. It can browse on workspaces and can create a child workspace to run a tool on it. **JaM**, **Join** and **Merge** workspaces, manages workspaces: creation and integration.

9 Conclusion

A*dage* is based on Unix and runs on SUN 3, SUN 4 under OS 3.4 or 4.0 and DEC Station 3100 under Ultrix. An A*dage* environment exists as soon as data are described: data storage, standard shell and graphical user-interface are ready to use tools. After that, the environment grows incrementally.

A first prototype exists, developed in 1987 and continued in 1988. This prototype validates the concepts and was used in the Esprit project METEOR. It is built upon a first version of the GDL, not powerful enough (no inheritance, no attributes, no node sharing...).

In 1988 has started the development of the current version of A*dage*. Presently exist PRESTO, L^3, $UTIL^3$, ADAGIO, GDL compiler, graal and the notion of hierarchical workspaces. We are currently working on a version of JaM integrating the derivation/lock model and we are studying the automatic derivation of actions to maintain constraints from a formal specification .

References

[1] S. Abiteboul, C. Beeri, *On the power of languages for the manipulation of complex objects,* Inria Research Report No 846, May 1988.

[2] E. W. Adams, M. Honda, T. C. Miller, *Object Management in a CASE Environment,* Proceedings of the 11th ICSE, Pittsburg, Pennsylvania, May 1989.

[3] Arvind, J.D. Brock, *Streams and Managers,* Proceedings of the 14th IBM Computer Science Symposium, 1983.

[4] M.P. Atkinson, O.P. Buneman, *Types and Persistence in Database Programming Languages,* ACM Computing surveys, Vol. 19, No. 2, June 1987.

[5] P. Bernas, M. Ferrario, *A Database Structure For Software Engineering Environment,* METEOR Technical Report t13/LRI/3-12, September 1987.

[6] P. Bernas, *The Pegase environment users's manual,* METEOR Technical Report t13/LRI/5, 1989.

[7] S. Ceri, S. Crespi-Reghizzi, G. lamperti, L. Lavazza, R. Zicari, *Algres : a system for the specification and prototyping of complex databases,* IEEE software 1988.

[8] P. Chen, *The Entity-Relationship Model: Towards a Unified View of Data,* ACM Transaction on Database Systems, January 1976.

[9] D. Clement, J. Heering, P Klint, J. Incerpi, G. Kahn, B. Lang, V. Pascual, *Preliminary of an environment generator,* GIPE CEC 348/A/T91/1,23.12.1986.

[10] K. R. Dittrich, W. Gotthard, P. C. Lockemann, *DAMOKLES - A Database System for Software Engineering Environments,* IFIP WG2.4 International Workshop on Advanced Programming Environnements, Trondheim, Norway, June 1986.

[11] A. Doucet, M.-C. Gaudel, *Bases de données et Génie logiciel: vers l'Intégration des Outils de Développement de Logiciel,* Journées Bases de Données de l'AFCET, La Rochelle, September 1986.

[12] M. Dowson, *ISTAR - An integrated Project Support Environment,* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Pratical Software Development Environments, January 1987.

[13] ECLIPSE, *ECLISPE - a Technical Overview,* Ref. 6329, Software Science, September 1987.

[14] G. C. Everest, *Object-Oriented DBMS,* Proceedings of the CASE' 88 Workshop, Cambridge, Massachussets, July 1988.

[15] T. Gallo, G. Serrano, F. Tisato, *ObNet: An Object-Oriented Approach for Supporting Large, Long-Lived, Highly Configurable Systems,* Proceedings of the 11th ICSE, Pittsburg, Pennsylvania, May 1989.

[16] J.Gettys, R.W.Scheifler, R.Newman, *Xlib-C Language X Interface, X Window System X.11.R3*, MIT, 1988.

[17] J.-L. Giavitto, Y. Holvoët, A. Mauboussin, P. Pauthe, *Guide-Lines for Building Adaptable Browsing Tools*, Esprit Technical Week, September 1987.

[18] J.-L. Giavitto, A. Devarenne, G. Rosuel, *Presto: des objets C++ persistants pour le système d'information d'Adage*, AFCET Journées d'Etudes Bases de Données Déductives et Bases de Données Orientées Objets, Paris, December 1988.

[19] J.L. Giavitto, A. Devarenne, G. Rosuel, Y. Holvoët, *Adage: New Trends in CASE Environments*, Proceedings of the International Conference on System Development Environments & Factories, Berlin, May 1989.

[20] Imperial Software Technology Ldt, *Requirements for Software Engineering Databases*, Final Report, Imperial College DoC, 1983

[21] P. Jeulin, P. Sauge, *GraphTalk : la Maitrise de la Qualité*, Génie Logiciel et Système Expert, December 1988.

[22] ISR, *Spokewindows Reference Manual*, Alacatel -I.S.R. 1989.

[23] A. van Lamsweerde, B. Delcourt, E. Delor, M.C. Schayes, and R. Champagne, *Generic Lifecycle Support in the ALMA Environment*, IEEE Transactions on Software Engineering, No 6 Vol 14, June 1988.

[24] D. E. Langworthy, *Object Oriented versus Relational Database Capabilities,* Proceedings of the CASE' 88 Workshop, Cambridge, Massachussets, July 1988.

[25] D. Maier, J. Stein, A. Otis, A. Purdy, *Development of an Object Oriented DBMS*, Proceedings of OOPSLA, Portland, Oregon, september 1986.

[26] C. Parent, S. Spaccapietra, *Gestion d'Objets Complexes avec des Entités Complexes*, AFCET Journées d'Etude Bases de Données Déductives et Bases de Données Orientées Objets, Paris, December 1988.

[27] PCTE, *PCTE, A Basis for a Portable Common Tool Environment, Functional Specifications*, Fourth Edition, NCC Ltd, November 1987.

[28] Sema-Group, *The Concerto Software Factory (general presentation)*, Sema-Group v1.4,1989.

[29] Sun, *Sunwindows reference manual*, Sun, 1988.

[30] Systematica, *The Virtual Software Factory*, Software Product Description, 1988.

[31] B. Stroustrup, *The C++ Programming Language*, Addison-Weslay, 1987.

[32] M. Verrall, *Tool Interaction and Integration in Software Engineering Environments,* Proceedings of the International Conference on System Development Environments & Factories, Berlin, May 1989.