Conceptual Modeling Using Conceptual Graphs

Guy W. Mineau
Dept. of Computer Science
Université Laval
Quebec City, Quebec
Canada, G1K 7P4
mineau@ift.ulaval.ca

Rokia Missaoui, Robert Godin
Dept. of Computer Science
Université du Québec à Montréal
Montreal, Quebec
Canada, H3C 3P8
{missaoui.rokia|godin.robert}@uqam.ca

Abstract

The next generation of task support systems aims at providing the relevant knowledge to knowledge-dependent activities, reducing the effort required to find or infer this knowledge from the data. In order to exploit knowledge associated with (or embedded in) databases and to migrate from data to knowledge management environments, conceptual modeling languages must offer more expressiveness than traditional modeling languages. This paper proposes a well established knowledge representation language, the conceptual graph (CG) formalism, as such a conceptual modeling language. It shows through an example, that it is suited for that purpose, allowing great expressiveness and uniformity of representation, though remaining simple to use. Its graphical interface and closeness to well-established modeling formalisms like UML and the ER model should facilitate its understanding and therefore should ease its integration into the market place. Therefore this paper emphasizes this aspect of the formalism.

1 Motivation

Databases were designed to support the data storage and retrieval activities related to information management. Because database management systems provide efficient task support to data intensive activities, tremendous gain in productivity was accomplished using these technologies. Furthermore, the information extracted from data often takes part in higher cognitive activities like strategic planning, process validation and decision making, which are key activities not only in the day-to-day operations of an organization but also in its medium and long-term development. Consequently, the next generation of task support systems aims at providing direct support to these knowledge-dependent activities.

This knowledge is generally related to different aspects of the management of an organization: its processes, customers, suppliers, markets, activities, past expertise, policies, guidelines, constraints, and so on. The storage, retrieval and processing of this type of information go way beyond available database technology; it calls for knowledge management systems.

In order to represent this type of information, traditional conceptual modeling languages are very limited due to their lack of semantic expressiveness. Therefore, in this paper, we propose a well established knowledge representation formalism, the conceptual graph (CG) formalism [1] as one such modeling language used to represent

the conceptual schema of a knowledge management system. Through an example, we show that the CG formalism has very attractive features such as higher expressiveness and deduction capabilities, though remaining simple to use, and providing a framework for executable specifications to be drawn and tested.
Because one of the most important features of the conceptual graph formalism is its graphical interface to a system of logic, we chose to compare it with another graphical model, UML [2], which is one of the most widely used modeling languages in the industry today. Of course other comparisons could be drawn. For instance, in [3] we compared the CG formalism with Telos, a description logic based formalism. In this paper, we rather put the emphasis on the graphical nature of the modeling constructs of the CG formalism. We think that this feature will eventually facilitate the integration of the formalism into the market place, and we therefore guided this presentation accordingly. Consequently, Section 2 gives a brief overview of conceptual modeling principles and introduces a simple example using UML. In Section 3 we present the CG version of the example and we summarize our point of view in Section 4.

2 Conceptual Modeling

A conceptual modeling language should allow the complete description of a domain. Usually, a domain is described by a set of objects which have relationships between them and which exhibit some behavior. The structure and behavior of objects as well as their relationships must comply with the semantic constraints of the domain.

2.1 The Structure of Objects

In Figure 1, we show a simplistic example of a university database. In UML, a solid-outline box is used to represent a *class* of objects (e.g., *Faculty*). A class abstracts the common structure and behavior of a set of objects. The box is usually divided into three compartments. The top compartment contains the class name and other general properties of the class. Classes can be grouped into *packages*. A path expression can also be used to prefix the class name in order to show from which package a class comes from. The middle compartment contains the list of *attributes* of the class. For example, the *Employee* class has four attributes: *SIN, position, salary,* and *bonus*. The bottom compartment contains a list of operations. For example, the *Student* class has the *averageMark* operation. Binary associations are represented by lines connecting two classes. For example the *Works-for* association represents the relationship between a *Faculty* and a *Department*.

2.2 The Behavior of Objects

Objects may have some behavior that impacts the world in which they take part. This behavior is useful to induce changes to a system. The declarative representation of behavior (of processes) could not seriously be considered until knowledge representation formalisms were made available.

In brief, behavior can be represented as processes which induce state changes on some system: data deletion, insertion or transformation, or as procedures which use the actual data for retrieval or inference purposes. In the past, processes were modeled in non-operational languages such as natural languages, or at best, were represented in programming languages. The advantage of the latter is that they could be executed, but in both cases, they were not integrated with the structural definition of the objects that they were applied on. The object-oriented paradigm provides a definition framework that allows structural and behavioral aspects of domain objects to be defined in an integrated framework. However, in this paradigm, the processes are still described in a programming language: no inferences on these processes are possible. We believe that there is an obvious need for a *declarative* representation of processes so that the information they convey becomes available to an inference engine, as are the structural definitions of domain objects. Having a declarative representation of a process is useful for making inferences in relation to the rest of the knowledge that is available to the inference engine. A process could then be simulated and validated and its impact could be foreseen. At this point in time where quality standards (ECI, ISO 9000+) are becoming more and more popular with government agencies and major corporations which require beforehand some form of quality control warranty, process validation plays a major role in key business activities such as assessing projects, bidding, obtaining contracts, ensuring quality control, and monitoring cost/benefit ratios, and consequently in the development of information systems. When task support is to be provided to higher cognitive activities that require to process knowledge inferred from the data, inferences from the processes involved may also be needed. So far, in today's information systems, this capability is lacking. Knowledge representation formalisms offer a solution to this problem, as discussed in [3,4] where CGs are used to render a uniform representation framework for both data and processes.

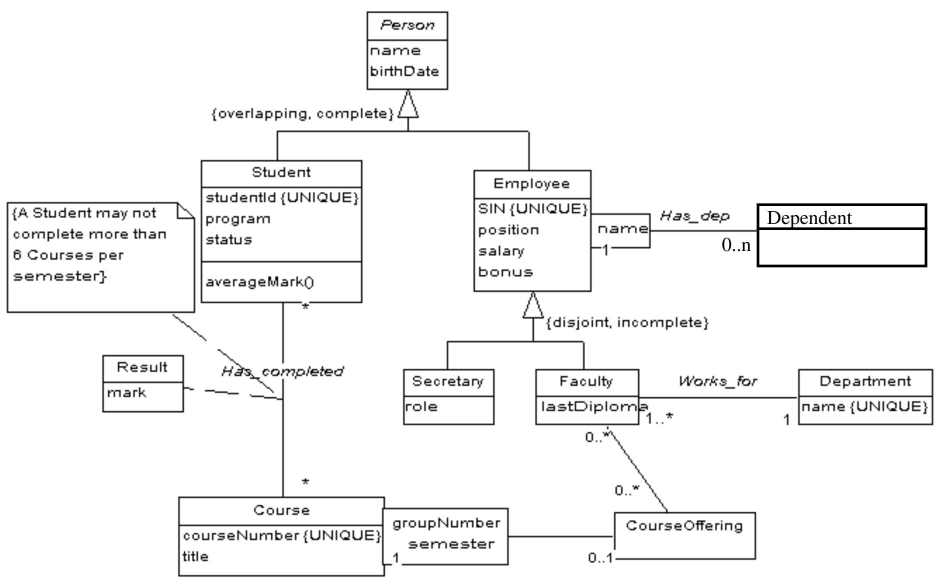

Figure 1. UML static structure diagram of our example domain.

2.3 Semantic Constraints

In traditional modeling languages, some semantic constraints can be expressed easily because the modeling language provides for their representation (e.g., cardinality constraints). However, most languages do not allow the representation of many useful constraints such as functional dependencies. Therefore, these constraints require some procedural implementation, i.e., some computer program that implements semantic integrity enforcement. The main problems with a procedural representation of constraints are: 1) they are hard-coded, which implies poor human comprehensibility and poor maintainability, 2) they cannot be used by the inference engine, which means little or no support pertaining to the explanation and validation (soundness) of the constraints themselves, and 3) they must be checked whenever a database modification operation occurs, which may require major processing resources, possibly preventing other applications from accessing the database meanwhile. Consequently, we advocate the use of a declarative representation for semantic constraints as proposed in some previous studies [5].

In UML class diagrams, a constraint is placed near the concerned elements or linked to them by dependency relationships. For instance, in our example we used the non standard convention of following the key attributes with the {UNIQUE} constraint. This works fine for single attribute keys. Other conventions should be used for multiple composite and possibly overlapping keys. For other more complicated constraints, the Object Constraint Language of UML (OCL), which is a predicate logic type language, must be used to express in a formal and precise way such constraints. Even though OCL aims at being used and understood by people that are not necessarily mathematicians [6], the application modeler needs to be comfortable with (fluent in) first order logic to correctly express and understand constraint formulation.

3 The CG Version of the Example

The conceptual graph (CG) formalism allows both structural (Section 3.1) and behavioral (Section 3.2) aspects of the definition of an object to be represented under the same graphical formalism [4]. Section 3.3 proposes a way to represent semantic constraints under the same graphical representation. We feel that by having only one single graphical representation framework for the representation of structure, behavior and constraints, the learning of the CG formalism and its subsequent integration to modeling activities should be facilitated.

3.1 Structural Descriptions in the CG Formalism

In the conceptual graph formalism [1], *types* of objects are defined. Types belong to a type lattice where a partial order of subsumption is defined on them. The only two predefined types are: T, the universal type (top element of the lattice), and $\perp$, the absurd type (bottom element). No individual is associated with the absurd type while all individuals are associated (either directly or through a subtype) with the universal type. Hence, if t_1 and t_2 are two different types, it is possible to know whether they are disjoint or if they overlap. If there exists a type t_3 different from the absurd type such that both types subsume t_3, then t_1 and t_2 are said to overlap. If their common subtype is the absurd type, then t_1 and t_2 are said to be disjoint. The type lattice represents the vocabulary of the domain.

Each individual is then associated with a type. The representation of an individual is called a *concept*. An individual may be known (represented by a constant) or unknown (then represented by an existentially quantified variable). In the former case, it is called an *individual* concept, while in the latter, it is called a *generic* concept. Graphically, a concept is represented by a box that contains the type of the individual, called *concept type*, and a reference to the individual (either constant or variable), called the *referent*. By default, a generic concept is existentially quantified even though the existential quantifier is not explicitly represented. However, it is possible to universally quantify a variable, producing a *universal concept* of some type. Figure 2 shows three concepts: the employee Peter (provided that "Peter" is a unique identifier in the system), an unknown employee (identified by an existentially quantified variable *x*), and a universally quantified concept representing any employee.

Relations are represented by labeled ovals connected to concepts through directed arcs. The direction of the arcs formally defines the order of the arguments of the relation, and helps produce an interpretation (usually in natural language) for the relation. Relations are N:M by definition, i.e., without cardinality constraint.

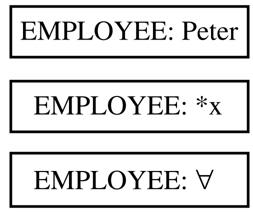

Figure 2. Three individuals of type EMPLOYEE.

Figure 3 shows the *has-dep* relation between an employee and a dependent. The interpretation of the graph of Figure 3 is: each employee has at least one dependent. This is not true in our example (see Figure 1) since the relation is optional.

In order to express optional relations in conceptual graphs, we must create a GUARDIAN type, subtype of EMPLOYEE, for which the relation would hold. So, according to Figure 1, the graph of Figure 4 should replace the one in Figure 3.

Figure 3. The representation of a N:M relation.

Figure 4. A specialization of the graph of Figure 3 with GUARDIAN < EMPLOYEE.

To deal with 1:1 relations, functions can be used. Functions were introduced in the CG theory by [7]. They are relations where the output concept (identified by the outgoing arc) is functionally determined by the input concepts of the relation (linked to the ongoing arcs). Functions are graphically represented as shown in Figure 5.

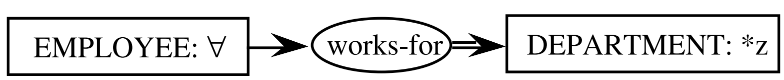

Figure 5. Example of a function.

In the CG formalism, attributes are independent objects linked through relations to the objects that they characterize. Unlike UML, no distinction is made between concepts and attributes: attribute values are considered to be individuals of some types used to describe other objects. The graph of Figure 6 shows part of the definition of a typical employee. This graph should be interpreted as: *each employee is a person uniquely identified by a SIN, who holds at least one position (but maybe more)*.

The problem with variables in first-order logic, like k and p in Figure 6, is that they represent unidentified values, e.g., null values. In some cases, null values may not be acceptable: e.g., primary key attributes. For instance, let us suppose that we want each employee to be given an individual SIN number as soon as it is created. Then, the SIN concept in Figure 6 must be individualized when a new employee is acquired. There was no way in the CG notation to enforce primary key constraints until [8] introduced some particular symbol #? in the referent field of a concept. The ? variable must be instantiated by a constant of the proper type as soon as the concept is asserted (that is, when a new employee is acquired in this case). The graph of Figure 6 can then be updated to use #? symbols wherever primary key constraints must be enforced (see Figure 7).

type EMPLOYEE(x) **is**:

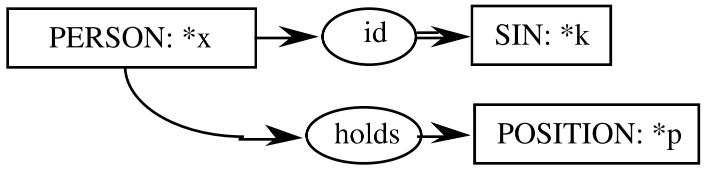

Figure 6. Part of the definition of a typical employee.

An individual must comply with the definition of the type it is associated with, but it may carry more information than required. For instance, Figure 8 shows the acquisition of employee Peter. This definition contains more information than required by the definition of a typical employee; this feature is not supported by UML.

type EMPLOYEE(x) **is**:

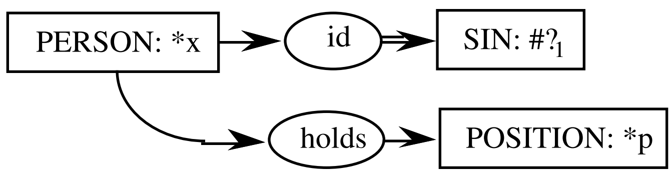

Figure 7. The new type definition for EMPLOYEE.

Relations must also be formally defined. They are described using a relation type only: they do not have referents. Figure 9 gives an example of the definition of the relation type *has-dep*. This definition is a graph that describes the relation between the concepts that it links. These concepts are said to be the formal parameters of the relation. The definition specifies the most general types for each parameter. All relations of that type used in conceptual graphs thereafter must be linked to concepts that are not more general than these formal parameters. For instance, with our example (provided that PARENT < PERSON and that FAMILY-MEMBER < PERSON) it requires that the two parameters of all relations of this type comply with these two more specific concept types; the relation could not be used to link arbitrary persons for instance.

individual EMPLOYEE(Peter) **is**:

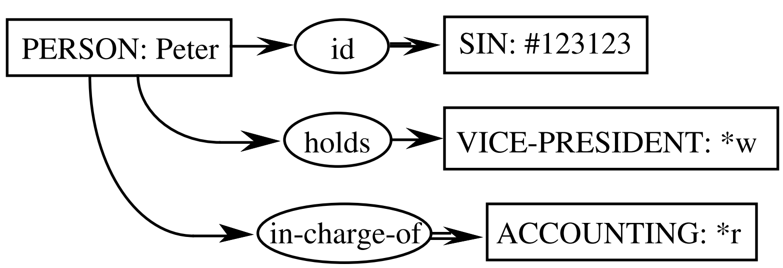

Figure 8. The definition of employee Peter.

relation has-dep(x,y) **is**:

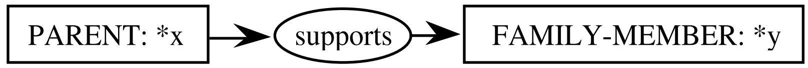

Figure 9. The definition of the relation type *has-dep*.

As shown in Figure 9, the *has-dep* relation type is a specialization of the *supports* relation, whose definition is given in Figure 10. The *supports* relation is defined using primitive relations such as *agnt*, *obj*, etc., i.e., relations defined as subtypes of the *link* relation type (the only predefined conceptual relation type).

Functions are defined exactly in the same way as relations are defined, except that the word *function* is used instead of *relation*. Basically, any relation can be used as a function by using a double arrow as the output arc.

relation supports(x,y) **is**:

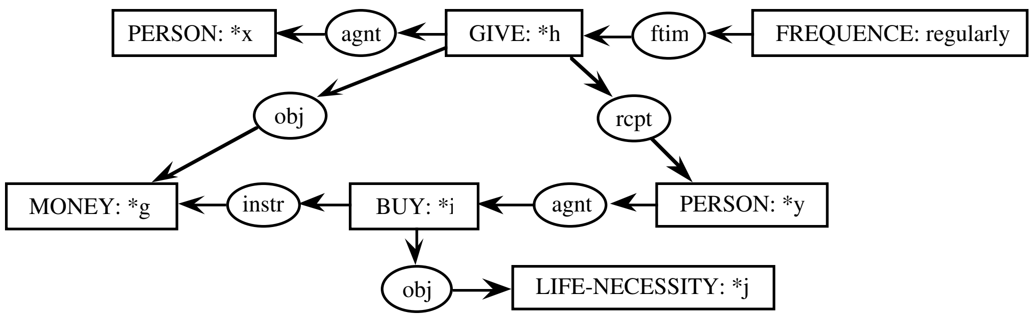

Figure 10. The definition of the relation type *supports*.

3.2 Behavioral Descriptions in the CG Formalism

The original theory [1] includes *actors*. Actors provide procedural attachments to CG systems. For particular computations, like with mathematical computations, actors provide a very efficient way to represent the relationships amongst members of infinite sets. Also, for applications that require on-the-spot computation, actors are useful because they produce an answer according to the actual state of the CG system (i.e., at the time the actor is triggered).

In a more general framework, the work presented in [9] introduces processes as pre/postcondition pairs. As such, processes are a generalization of actors; they allow arbitrary graphs to be asserted or negated. In the following, we briefly present actors and processes.

3.2.1 Actors

Actors are useful to compute the value of some concept, i.e., they individualize a generic concept to the appropriate individual value, according to its associated input parameters. For instance, let us suppose that we use the actor definition mechanism as illustrated in Figure 11. The actual procedure used for computing the sum of two integer values will be associated with the *sum* symbol used here. Figure 12 shows how this actor could be used in a conceptual graph to achieve this computation. As soon as all the input concepts of the actor are instantiated, i.e., are known to be identifiable individual objects, this triggers the computation which then produces an output value, instantiating variable *f* to this value. With our example, when we know the exact number of overhead pages (called initial pages) required in the preparation of a paper: title page, cover page, etc., and the actual number of pages for the content of the paper, automatically the total number of pages is computed and asserted as part of the knowledge of the system.

actor sum(**in**$_1$ x; **in**$_2$ y; **out** z) **is**:

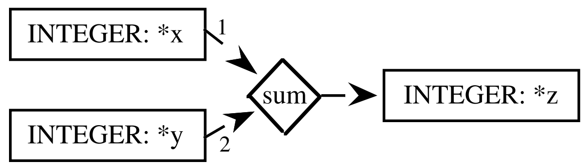

Figure 11. The definition of the actor *sum*.

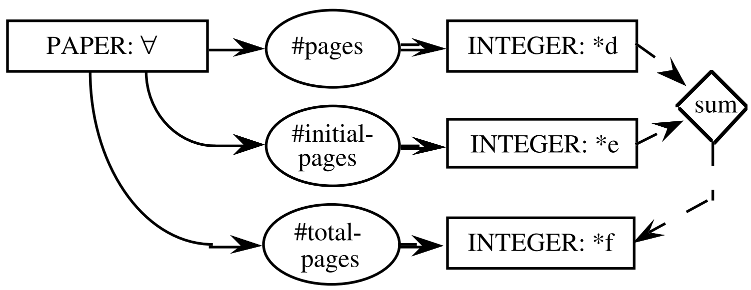

Figure 12. Using an actor in a graph: binding it to some concepts.

3.2.2 Processes

In the CG literature today there are some proposals to represent processes in a declarative form based on the CG formalism [10]. In this section we present the approach of [9] in which a process is represented as a set of transitions from pre to postconditions. All pre and postconditions are represented as simple conceptual graphs. However, a transition is represented using *nested graphs*, i.e., graphs in which some concepts have conceptual graphs as referents. Nested graphs allow the description of meta-level knowledge. We chose to present this approach because it is fully compatible with the representation of constraints as presented in Section 3.3 below, giving a unified representation framework for the various modeling activities needed to describe some knowledge domain, i.e., to describe structure, dynamics and constraints on both.

[9] shows how to translate well-defined processes into pre/postcondition pairs. For any process representation language that is based on a state transition machine, there is always a translation possible toward pre/postcondition pairs. It is thus possible to use some programming language (such as C or some visual programming language) to describe a process (see Figure 13), and then have it translated into pre/postcondition pairs in order to use it in an inference engine, ensuring its full integration with the rest of the CG system.

In this approach, each possible transformation is described as a transition that occurs when preconditions are met, producing postconditions. In our example, the synchronization structure is predicated and integrated to the pre and postconditions. With the process of Figure 13, the objects are variables; the synchronization structure is based on the dependency between variable updates and use. Figure 14 shows the synchronization structure (graph) of the process of Figure 13. Figures 15 and 16 show the two transition rules describing line l_5 of Figure 13 (which is a variable assignment instruction). A transition rule is a pair of associated pre and postconditions. It can be seen as a subprocess. Of course, [9] covers all algorithmic constructs found in programming languages. The synchronization graph of Figure 14 and the transition rules of Figures 15 and 16 could be produced automatically from the analysis of the process of Figure 13. The point that we are trying to make here is that process modeling could be done using any high-level programming-like process modeling language at the knowledge engineer's convenience. The end-result would be nevertheless a full integration of the described processes with the CG base, in the same declarative format, making the knowledge that they convey available to the inference engine and subject to constraint validation (as explained in Section 3.3 below).

```
l0:   int fact(int n)
l1:   {  int temp;
l2:      int ftemp;
l3:      int f;
l4:      if (n == 0 or n == 1)
l5:            f = 1;
l6:      else
l7:      {     temp = n - 1;
l8:            ftemp = fact(temp);
l9:      f = n * ftemp;
l10:     }
l11:     return f; }
l12:  }
```

Figure 13. A simple process: the recursive factorial function (written in C).

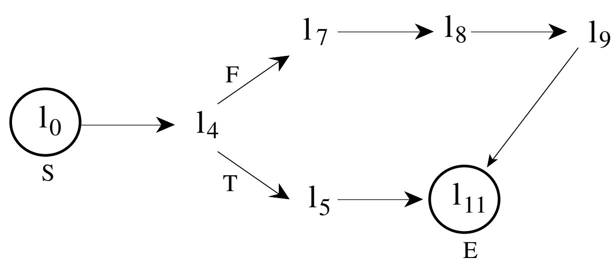

Figure 14. The synchronization graph of the process of Figure 13.

The graphical representation of a process is a double-lined diamond box linked to statement boxes, one for each parameter (see Figure 17). The precondition boxes are labeled CONDITION or NEGATION (with CONDITION < STATEMENT and NEGATION < STATEMENT) depending if the condition to be verified must have a TRUE or FALSE value. Also, the postcondition boxes are labeled ASSERTION or NEGATION depending if the graph it contains as referent must be asserted or negated. A process is triggered when all its precondition graphs are triggered (instantiated), then all its postcondition graphs are either asserted or negated. Of course, if an asserted (negated) graph is already known to be true (false), the update has no effect on the state of the system with regard to that operation.

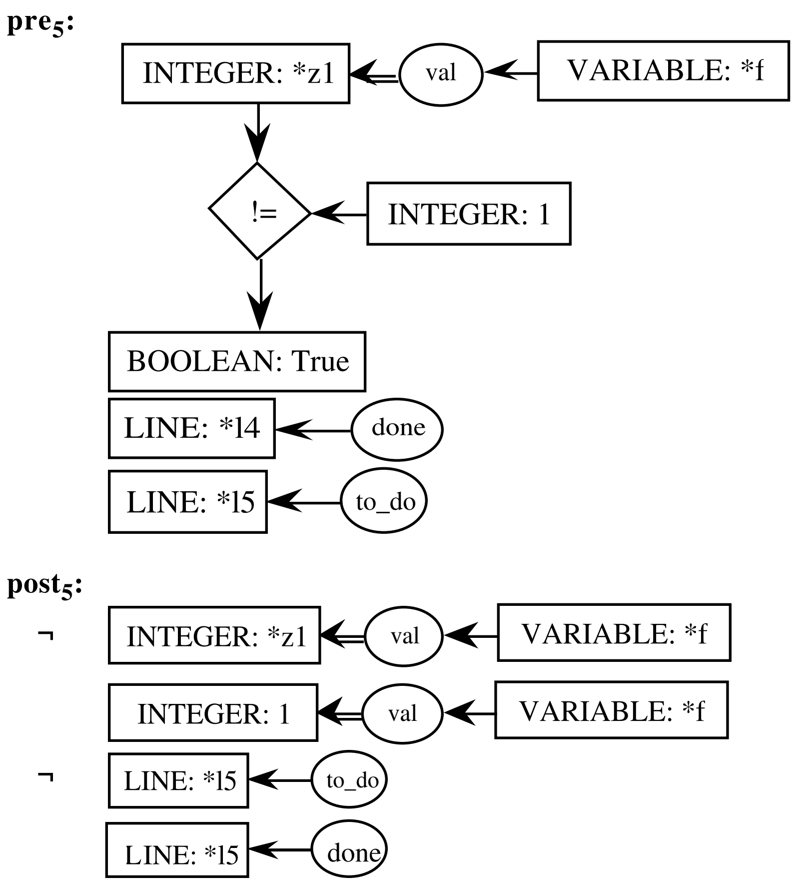

Figure 15. The transition rule corresponding to the initial state where $f \neq 1$.

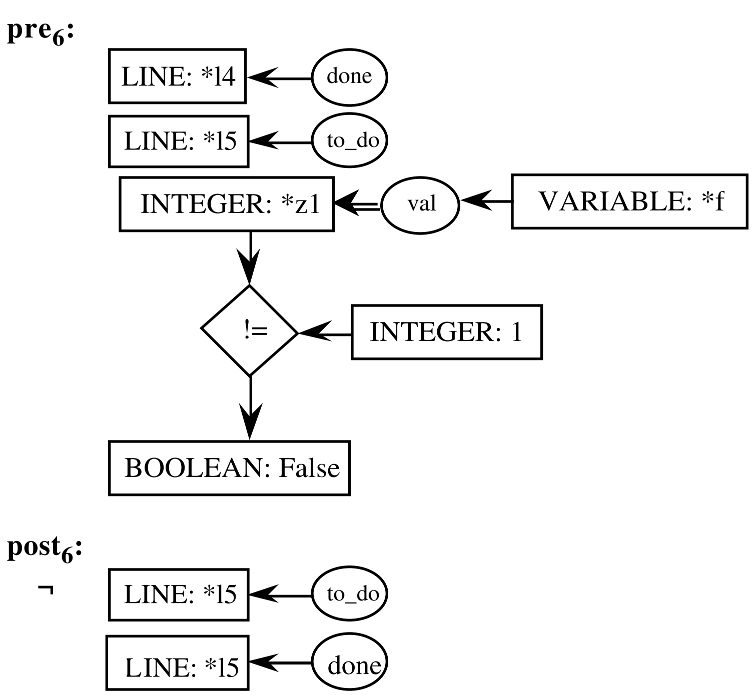

Figure 16. The transition rule corresponding to the initial state where $f = 1$.

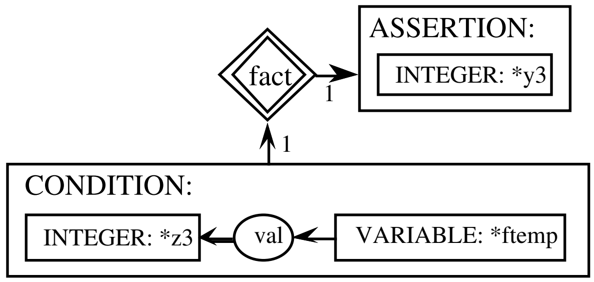

Figure 17. The graphical representation of a process.

With our example, the factorial process would be triggered by asserting the graph of Figure 17. The answer, the concept [INTEGER: *y3], would be available to the whole CG system through the y3 variable which can be referenced in any other graph after the process terminates. Process calls, parameter passing and execution monitoring are described in [9].

3.3 Semantic Constraints in the CG Formalism

Constraints are of the outmost importance; the validity of the model greatly depends on them. They prevent data manipulation from producing and/or storing facts known to be false. They must be enforced for each state of the data (knowledge) base. Traditionally, they are hard-coded and each update or retrieval operation triggers a procedure that enforces all the relevant constraints. However, procedural representations have their limitations, and a *declarative* representation should be sought as motivated earlier.

When modeling real applications, all sorts of semantic constraints on the structural definition of an object must be represented. For instance, the basic UML constructs give us some constraint representation: cardinalities for relations (optional or mandatory status), complete or partial subsumption, set intersection (overlapping or disjoint sibling types). Unfortunately, there are many constraints that can not be represented in UML (unless a FOL-based language is used). They would be useful for correctly modeling an application domain, and thus, for improving the validity of the conceptual schema, especially when a computational model is derived from it.

Behavioral aspects of a definition also need to be constrained. For instance, no graduate student can lose its graduate status and become an undergraduate student again. In that case, any state transition that updates the knowledge base by deleting the graduate status of any graduate student should be blocked. Processes represent data manipulation operations. These operations must comply with the semantics of the application domain as well.

The main idea in the framework that we summarize here, originally proposed in [8], is to identify subspaces of sentences which should not be stored in the knowledge base because we know that they violate some semantic constraint. By doing so, we aim at excluding false facts. For instance, let us assume that no employee should have more than two dependents (which is a cardinality constraint). In that case, we must negate any graph which states that there is an employee who has (at least) three dependents. Figure 18 shows how this constraint would be represented. Here we use the keyword **false** to define that constraint.

false:

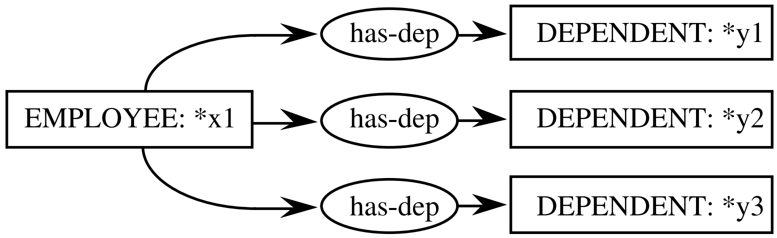

Figure 18. A first semantic constraint.

Any specialization of the graph of Figure 18 makes it true while in fact it should be false (as stated by the constraint).[1] Since a constraint makes sure that no graph more specialized than the graph appearing under the **false** clause is stored, the constraint represents a *subspace* of graphs, of sentences expressible in the description language, all in violation of some semantic constraint. Let us now assume that the constraint does not hold if the employee is remarried (in which case he may have more than two dependents). Then we would need to relax the constraint of Figure 18 by adding an exception to it. We achieve this by adding an **unless** clause as shown in Figure 19. This exception allows a graph that is a specialization of both Figure 18 and Figure 19 to be stored in the system, i.e., to be considered valid.

unless:

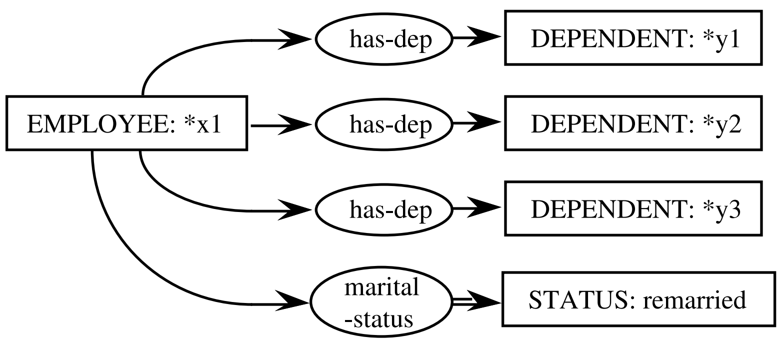

Figure 19. An exception to the constraint of Figure 18.

As for constraints applicable on processes, one must represent the transition rule to be constrained as a sentence of the description language, and have a constraint described by a **false-unless** clause associated with it. For example, we must express the transition constraint of Figure 20 to state that: *no process can delete the graduate status of a graduate student*. Similarly to what was explained above, any transition rule that would be a specialization of Figure 20 could not be asserted (executed) because it would violate this constraint.

In brief, all semantic constraints normally found in database literature can be represented using what is proposed in this section. The CG formalism thus offers a *graphical* and *uniform* representation for structure, behavior and constraints. Once again we strongly advocate the use of a single and simple conceptual modeling language for all aspects of an application domain, which allows their full integration and their availability to an inference engine.

false:

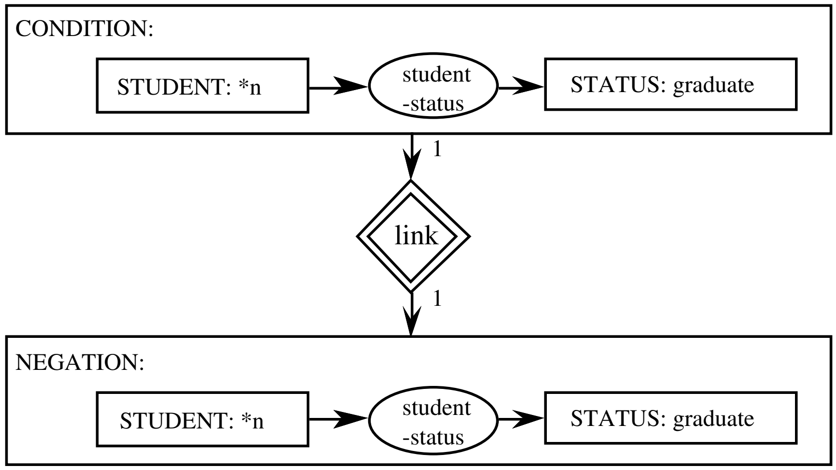

Figure 20. Expressing a constraint on a process.

4 Discussion

The objective of this paper was to outline the potential of the CG formalism for conceptual modeling for both database and knowledge base systems. To reach our goal, we presented the main features of the CG formalism through a simple example. We come to the conclusion that the CG formalism offers simple yet powerful constructs to model all aspects of an application domain in a *totally integrated* manner. The disadvantages associated with using the CG formalism are mainly due to its rigidity (i.e., it has formal semantics), to the lack of a commercially available workbench (though many prototypes exist), and to the fact that the global view of a domain as usually represented with UML diagrams is broken down into many small conceptual graphs. However, small CGs could be joined to produce a global overview of a domain (producing diagrams like UML's).

When developing a conceptual model, because of the precise semantics of the CG formalism, ambiguities could be detected and the application modeler could be prompted to resolve them. Based on our fruitful experience with an industrial partner, we strongly believe that simple yet formal representation theories like the CG formalism could be used in specification engineering. The informal cognitive task associated with the elicitation of a conceptual schema could be supported by the formal deduction engine of the CG system (including constraint compliance). Obviously, this opens the door to further research in that direction.

The CG formalism is one of few logic-based modeling languages allowing easy description of a domain into executable specifications, in a totally integrated manner, using a graphical interface. Its main strength is that it bridges the gap between traditional conceptual modeling languages and more advanced models. Its graphical features, much like those of UML and the ER model, rapidly become easy to manipulate. Mappable to FOL, it offers to incorporate an inference engine to any task support facility. Close to the ER model, it is easy to implement.

Finally, any conceptual model unambiguously defined can be translated into a CG-based model. This adds new functionalities to the system, like inference capabilities, the automatic generation of explanations (needed for task support), and other functionalities related to knowledge management environments, thus bridging the gap between data and knowledge base systems. Therefore, we advocate its use.

References

[1] Sowa, J.F., (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley.

[2] Booch, G., Rumbaugh, J.E. & Jacobson, I., (1999). *The Unified Modeling Language User Guide,* Addison-Wesley.

[3] Mineau, G.W., Missaoui R. & Godin, R., (2000a). Conceptual Modeling for Data and Knowledge Management. *Data and Knowledge Engineering*, **33**(2). 137-168.

[4] Mineau, G.W., Missaoui, R. & Godin, R., (2000b). Conceptual Modeling in the Knowledge Management Era, Technical Report, UQAM, March 2000.

[5] Di Battista, G. & Lenzirini, M., (1993) Deductive Entity Relationship Modeling. *IEEE Transactions on Knowledge and Data Engineering*, **5**(3). 439-450.

[6] Warmer, J. & Kleppe, A., (1999). *The Object Constraint Language - Precise Modeling with UML*. Addison-Wesley.

[7] Mineau, G.W., (1994). Views, Mappings and Functions: Essential Definitions to the Conceptual Graph Theory. Lecture Notes in AI #835. W.M. Tepfenhart, J.P. Dick & J.F. Sowa (Eds). Springer-Verlag. 160-174.

[8] Mineau, G.W. & Missaoui, R., (1997). The Representation of Semantic Constraints in CG Systems. Lecture Notes in AI #1257. D. Lukose, H. Delugach, M. Keeler, L. Searle & J.F. Sowa (Eds). Springer-Verlag. 138-152.

[9] Mineau, G.W., (1998). From Actors to Processes: The Representation of Dynamic Knowledge Using Conceptual Graphs. Lecture Notes in AI #1453. M.-L. Mugnier & M. Chein (Eds). Springer-Verlag. 65-79.

[10] Lukose, D. & Mineau, G.W., (1998). A Comparative Study of Dynamic Conceptual Graphs. *Proc. of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-98).* B. Gaines & M. Musen (Eds). University of Calgary, Alberta, Canada. Section VKM-7. 20 pages.