GRAPHICAL REPRESENTATION OF RULES IN AN OBJECT-ORIENTED ENVIRONMENT

Kenneth Messa
Bogdan Czejdo
Erick Villalobos

Department of Mathematics and Computer Science
Loyola University
New Orleans, LA

ABSTRACT

In this paper a graphical representation of rules is proposed. The graphical rules are formulated based on Extended Entity-Relationship (EER) diagrams. Such diagrams are compiled into an object-oriented language code that is included as a component of the developed software system. Other components can be directly coded in an object-oriented language or generated using other graphical tools. We identify and discuss two types of graphical rules: class graphical rules and instance graphical rules. The syntax and semantics of both types of graphical rules are discussed. The graphical rules can contain object-oriented code in addition to graphical diagrams. The methods of integration of diagrams and object-oriented code are described.

1. INTRODUCTION

In the process of development of many software systems, including interactive graphical systems [2], experiments with a variety of working models are generally suggested. Rapid prototyping can be very helpful in the development process of such systems [14]. Rule-based programming [1, 5, 6, 12] is considered a good candidate for rapid prototyping because of the relatively simple procedure of inserting and deleting rules for the prototyped system. At the same time, existing rule-based programming languages do not support encapsulation and modularity and, as a result, it seems quite difficult to develop large rule-based systems [13]. One way to take significant advantage of rule-based programming is to integrate the rule-based system with an object-oriented language [10] or database language [7].

This research was partially supported by NSF grant # CDA-9300071

In this paper we describe an object-oriented environment that includes database primitives and show how graphical rules can be integrated into such an environment. The approach is an extension of our previous work [3, 4, 8, 9], where we proposed the use of graphical specifications based on an extended Entity-Relationship (EER) model and showed how such graphical specifications can be described in an object-oriented environment.

We identify and discuss two types of graphical rules: class graphical rules and instance graphical rules. Specification of each type of graphical rule can contain both graphical diagrams and object-oriented code. The syntax of graphical diagrams is defined by an EER meta-model. The semantics of graphical diagrams is described by the graphical rules themselves (meta-rules). A Smalltalk database library is assumed for the parts requiring direct object-oriented code.

The paper is organized as follows. In the next section, the description of two types of graphical rules is provided. In Section 3 the processing of graphical rules is discussed. In Section 4 the syntax of graphical rules is defined using an EER meta-model. The semantics of graphical rules is discussed in Section 5. The Summary presents some conclusions.

2. CLASS AND INSTANCE GRAPHICAL RULES

In this section we describe class and instance graphical rules. We will give an example of each type of graphical rule and describe how it is processed.

A *class rule* is a rule that is applied to a class and consists of three elements: base class name (CN), left hand side (LHS) and right hand side (RHS). The LHS consists of a non-empty sequence of condition components. Each condition component is either a condition diagram (CD) or condition text (CT). The CD and CT components can share the same variables. The LHS component returns a single boolean value which determines if the RHS is to be executed. The RHS component describes actions that are executed when the rule is fired. These actions consist of a sequence of action components. Each action component is either an action diagram (AD) or action text (AT). In order to integrate efficiently these components, we allow them to share the same variables.

An *instance rule* consists of three components: base class name (CN), left hand side (LHS) and right hand side (RHS). The structure of the LHS of an instance rule is the same as the structure of the LHS of a class rule. The LHS component of an instance rule returns a boolean value for each object in the CN class. This boolean value determines if the RHS is to be executed for a particular object. The RHS component for an instance rule is also similar to the RHS component of a class rule. The only difference is that here it is executed for each applicable object.

Both condition diagrams and action diagrams are based on Extended Entity-Relationship (EER) diagrams [3, 9]. As an example let us consider the data processing system for student registration that includes two entity sets Student and Section, and two relationship sets Enrolled and Waitlisted.

The entity sets are represented graphically, as in typical ER diagrams, by a rectangle containing the name of that entity set. The relationship sets are represented graphically by a diamond box containing the name of that relationship set.

The selection conditions are indicated by the condition box. Conditions can be of the following type: *attribute condition* and *set condition.* Attribute conditions are represented graphically by attaching the condition box to an attribute icon. Set conditions are represented graphically by attaching the condition box to an entity set icon.

Variables can be of the following type: *attribute variable,* and *set variable.* Attribute variables are represented graphically by attaching the variable box to an attribute icon. Set variables are represented graphically by attaching the variable box to an entity set icon.

Let us consider a rule:

Rule 1: "If there are less than 5 computer science majors, then change their major to 'Math' and list their names".

This is the class rule because the LHS component is based on a class and can be rewritten as "If there are less than 5 computer science majors in the class Student...". The LHS components can be represented in Figure 1A (condition diagram) and Figure 1B (condition text). In the condition diagram, the set of students who satisfy the condition is selected and the result is stored in the variable labeled VARa.

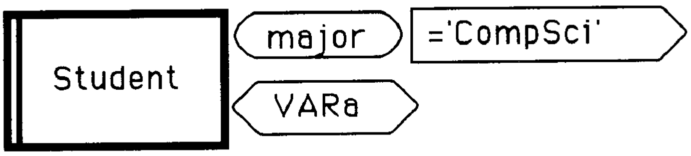

Figure 1A. Condition Diagram for **Rule 1**.

The boolean value is computed for the condition text by comparing the count of VARa with the number 5. The operator *count* is defined for all sets of objects.

^VARa count < 5

Figure 1B. Condition Text for **Rule 1.**

The RHS of the rule 1 can be specified by an action diagram in Figure 1C and action text in Figure 1D. In the action diagram, all students who are currently Computer Science majors are identified, their names are stored in the set variable VAR1 and the property 'major' takes the new value 'Math'. In the action text the appropriate I/O operations are performed, i.e. the names of identified students are printed.

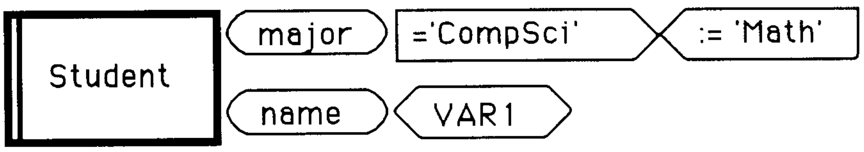

Figure 1C. Action Diagram for **Rule 1**.

VAR1 list.

Figure 1D. Action Text for **Rule 1**.

Let us consider another rule:

Rule 2: "If any section has at most 3 students on the waiting list, then change their status to enrolled for that same section" .

This is an instance rule because the LHS describes a condition for each object in the class Section. The LHS components can be represented in Figure 2A (condition diagram) and Figure 2B (condition text). In the condition diagram, the set of students who are on the waiting list for a given course is selected and the result is stored in the variable labeled VARa.

The boolean value is computed using the condition text by comparing the count of VARa with the number 3.

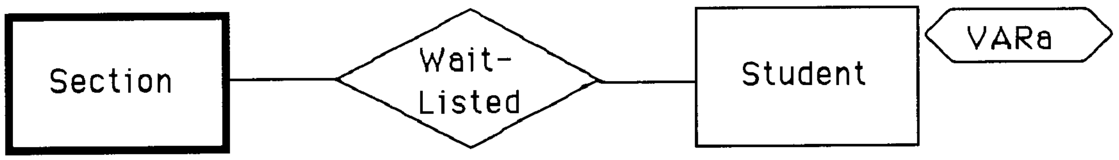

Figure 2A. Condition Diagram for **Rule 2**.

^VARa count <= 3

Figure 2B. Condition Text for **Rule 2**

The RHS of the rule 2 can be specified by two action diagrams in Figure 2C and 2D. In the first action diagram, all students who are currently on the waitlist for the given section are identified, placed in VAR1 and then deleted from waitlist. The deletion is indicated by filling the icon of the relationship set WaitListed with characters "D". In the second action diagram, each student in VAR1 is enrolled into particular section The insertion is indicated by filling the icon of the relationship set Enrolled with characters "I". Since the RHS of this rule can be specified completely by the action diagrams, there is no action text for this rule.

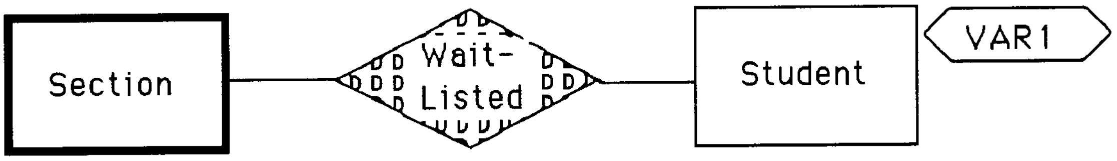

Figure 2C. Action Diagram 1 for **Rule 2**.

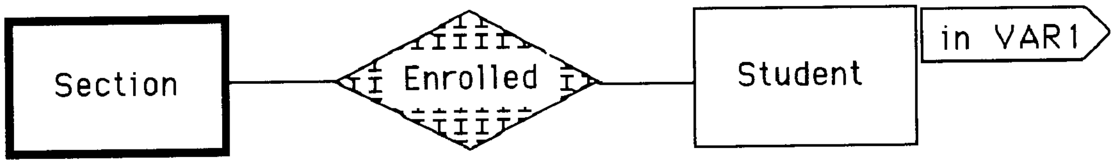

Figure 2D. Action Diagram 2 for **Rule 2**.

3. PROCESSING OF GRAPHICAL RULES

The graphical rules can be translated into object-oriented code (e.g. into Smalltalk code) and executed using an inference engine. In order to simplify the translation we will assume that the object-oriented programming environment will contain a library of database abstractions. In this section we will shortly describe this library. Then we will show how graphical rules are translated using Rules 1 and 2 as examples.

3.1 Library of Database Abstractions

The Library of Object-oriented Database Abstractions (LODA) was written by us in the object-oriented language Smalltalk [10], but can be easily expressed in other object-oriented programming languages. The library is based on Extended Entity-Relationship model abstractions.

Specification of queries in an EER database involves use of the *selectWhere, selectAll, relate* and *relateUsing* operators. The selectWhere message selects those objects from the set that satisfy some condition. For instance, in order to select all faculty whose degree is PhD, we might invoke the message

```
Faculty selectWhere:#('degree' '=' 'PhD').
```

This operator builds a new set from an existing one with a (usually) smaller collection of objects.

The selectAll message selects all of the objects from the set. For instance, we can select all faculty by invoking the message

```
Faculty selectAll.
```

This operator also builds a new set from an existing one.

In order to relate objects using a specific relationship set, one specifies the entity set and the connecting relationship set. For instance, we may want to relate the entities in Faculty using the relationship Teaches to obtain a subset of Student. We would call this operation by:

```
Faculty relateUsing:Teaches.
```

Another relationship is defined using the subclass/superclass relationship between classes. The *intersect:* operator allows one to obtain the set of entities that are members of a subclass. For example, if TenuredFaculty is a subclass of Faculty, and if setOfFaculty is a set of Faculty entities, then the operator below returns the set of tenured faculty that are in setOfFaculty:

```
setOfFaculty intersect:TenuredFaculty.
```

To embed a set of entities from the subclass into the parent class, a simple assignment operator is used.

In order to retrieve the values of an attribute, one specifies the entity and the attribute name. For instance, we may want to retrieve the value of 'degree' of an entity aFaculty by the following operation:

```
aFaculty degree.
```

This operation can also be applied to an entity set in a "projection" operation. For instance, in order to retrieve the value of 'name' of each entity in the set setOfFaculty, we invoke the following:

```
setOfFaculty name.
```

In addition to actual attributes, there are certain additional "calculated" attributes that are available for entity sets. For example, one may determine the number of entities of the particular entity set, setOfFaculty using the operator *count* as in:

```
setOfFaculty count.
```

This operator may be invoked for the entire entity set as well.

We also need the facility to insert and delete data. In our object-oriented system, in order to insert values into an entity set, the messages insertValues: and deleteWhere: are used to insert and delete data respectively.

The *changeAll:with:* operator updates the value of the indicated attribute and might be invoked as in the example:

```
setOfFaculty changeAll:'degree' with:'PhD'.
```

Similar operations are available for the relationship sets.

Operations are also available to create and destroy entity sets as part of a database. For creation, there is an operation *createEntitySet:withAttributes:* that creates an entity set with particular attributes and adds it to the database and the message to remove an entity set from a database is given by *dropEntitySet:*. There also are messages to create and destroy relationship sets. All of the operators are described in more detail in [8].

3.2 Translation of Graphical Rules

Graphical diagrams that are part of graphical rules are translated into object-oriented code that uses the library of database abstractions. As an example of this conversion, let us consider the condition diagram of the graphical rule in Figure 1A, which can be converted and appended to the condition text giving as a result the following:

```
setOfStudent :=
    Student selectWhere:#('major' '=' 'CompSci').
VARa := setOfStudent.
^VARa count< 5
```

The action diagram of this rule (from Figure 1C) can be converted and appended to the action text giving as a result the following:

```
setOfStudent :=
    Student selectWhere:#('major' '=' 'CompSci').
setOfStudent changeAll:'major' with:'Math'.
VAR1 := setOfStudent name.
VAR1 list.
```

As a second example, let us consider the condition diagram of the graphical rule in Figure 2A, which can be converted and appended to the condition text giving as a result the following:

```
setOfStudent := object relateUsing:WaitListed.
VARa := setOfStudent.
^VARa count <= 3
```

In this code, *object* refers to each object in Section. Sections that satisfy the LHS will be passed to the RHS by the inference engine, as described in the next section.

The action diagram of this rule (from Figures 2C and 2D) can be converted, giving as a result the following:

```
setOfStudent := object relateUsing:Waitlisted.
```

```
VAR1 := setOfStudent.
  Waitlisted delete:object and:setOfStudent.
  setOfStudent := Student select:VAR1.
  Enrolled newWith:object and:setOfStudent.
```

3.3 Inference Engine

The graphical rules need to be processed in a way similar to other rule-based systems. Here we describe a simplified inference engine that shows the basic method for rules execution. More advanced inference engines can also be used, such as the one described in [5].

The operator *apply:* determines the application of the rules. This operator applies the rules in a pre-determined order until no left hand side of any rule is satisfied. The *apply:* operator also checks that each rule is not executed twice in the same environment.

```
Rules apply:[:aRule| aRule fire]
```

In the case of a class rule, the fire operator can be defined as follows:

```
aRule perform:(aRule lhs) with:(aRule className) ifTrue:[
      self perform:(aRule rhs) with:(aRule className)].
```

In this definition aRule represents the rule that is being fired. The message *lhs* returns the LHS component of the rule, the message *className* returns the class name of the rule and *rhs* returns the RHS component.

In the case of an instance rule, the operator fire is defined as:

```
aRule className do:[object|
  aRule perform:(aRule lhs) with:object ifTrue:[
      aRule perform:(aRule rhs) with:object.]].
```

This fire operator enumerates through all objects of the base class (className) and executes the RHS for those objects where the LHS evaluates to true.

4. SUMMARY

In this paper a graphical representation of rules is proposed. The graphical rules are formulated based on Extended Entity-Relationship (EER) diagrams. Such diagrams are compiled into an object-oriented language code that is included as a component of the developed software system.

We identified and discussed two types of graphical rules: class graphical rules and instance graphical rules. The syntax and semantics of both types of graphical rules were discussed. We allow both graphical diagrams and object-oriented code in each part of the graphical rule. This way, we significantly alleviate the mismatch between the rule-based programming paradigm and the object-oriented paradigm.

5. REFERENCES

1. L. Browson, et al. *Programming Expert Systems in OPS5--An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985.
2. S. Chang (ed). *Visual Languages*, Plenum Press, NY, 1986.
3. B. Czejdo, R. Elmasri, M. Rusinkiewicz and D. Embley. "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model", *IEEE Computer*, March 1990.
4. B. Czejdo and K. Messa. "Generating Smalltalk Code From Graphical Operations and Rules", *Proceedings of the IEEE Symposium on Visual Languages*, Bergen, Norway, 1993.
5. C. Eick. "Activation pattern controlled rules: towards the integration of data-driven and command-driven programming", *Applied Intelligence,* 2(1):75-91, 1992.
6. J. Giarratano and G. Riley. *ExpertSystems -- Principles and Programming*, PWS-Kent, Boston, 1989.
7. A. Hsu and T. Imielinski. "Integrity checking for multiple updates", *Proceedings of the ACM-SIGMOD Conference on Management of Data*, Austin, 1985, pp. 152-164.
8. K. Messa and B. Czejdo. "Entity-Relationship Model in Object Oriented Languages and its Applications", *Proceedings of the 1992 Annual Conference of the International Association For Computer Information Systems*, New Orleans, 1992, pp. 301-311.
9. K. Messa and B. Czejdo. "Generating Database Applications in Smalltalk from an Extended Entity-Relationship Model", *Proceedings of the 17th Annual Energy-sources Technology Conference*, New Orleans, January, 1994.
10. D. Miranker, et al. "The C++ embeddable rule system", *Proceedings of the International Conference on Tools for AI*, San Jose, November 1991, pp. 386-
11. Objectworks\Smalltalk Release 4.1. *User's Guide, ParcPlace Systems*, Sunnyvale, California, 1992.
12. N. Rowe. *Artificial Intelligence Through PROLOG*, Prentice Hall, Englewood Cliffs, NJ, 1988.
13. E. Soloway, J. Bachant and J. Jensen. "Accessing the maintainability of XCON-in-RIME coping with problems of very large rule-bases", *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, 1987, pp. 824-829.
14. M.M. Tanik and R.T. Yeh. "Rapid Prototyping in Software Development", *Computer*, May, 1989.